Information Systems Vol. 15, No. 1, pp. 85–116, 1990
Printed in Great Britain.

0306-4379/90 $3.00 + 0.00

A SOFTWARE PROCESS DATA MODEL FOR KNOWLEDGE ENGINEERING IN INFORMATION SYSTEMS

MATTHIAS JARKE, MANFRED JEUSFELD and THOMAS ROSE
University of Passau, P.O. Box 2540, D-8390 Passau, F.R.G.

(*Submitted March* 1989; *Received for publication* 18 *October* 1989)

Abstract—Knowledge engineering for information systems is a long-term, multi-person task that requires tight control and memorization not only of *what* knowledge is acquired but also of *why* and *how* it is acquired. We propose a software process data model as the foundation of a knowledge-based software information system that emphasizes control, support and documentation of design decision-making and tool integration in information systems environments.

The model is developed along two dimensions. Firstly, it defines how to represent and integrate design objects (what), design decisions (why) and design tools (how). Secondly, it exploits the abstraction mechanisms of the extensible hybrid knowledge representation language CML/Telos to manage the evolution not only of particular software projects, but also of the software development environment in which these projects operate. Modular aggregation relates design-in-the-small and design-in-the-large support. Besides motivating and formalizing the model, we describe an operational prototype implementation called *ConceptBase* and report intitial application experiences in the DAIDA ESPRIT project.

Key words: Software databases, software process models, information systems engineering, knowledge base management systems.

1. INTRODUCTION

Knowledge engineering has been publicized as a technology to build and maintain the knowledge base of so-called *expert systems*, systems intended to mimick the performance of human experts in specialized domains of diagnosis, design, medical and business decision support, etc. An expert system uses a narrow set of specialized algorithms, the "inference engine", to work on a generalized data structure or "knowledge base" that represents the expert's domain knowledge and problem-solving strategies. Expert system "shells" have evolved as a technology to support knowledge engineering but knowledge engineering has also been considered as a new kind of human profession similar to software engineering.

While the last few years have seen strong interest in integrating knowledge-based systems and information systems technologies [1], the relationships between knowledge *engineering* and information systems have captured less attention. One way to address this problem is to view expert systems development as a special case of information systems development in which the target software environment (an expert systems shell) offers richer data structures and different kinds of processing methods. In particular, rapid prototyping, expert knowledge consistency checking and evolution support are often emphasized in expert systems development methodologies.

In this paper, we shall be more interested in another way of relating knowledge engineering with information systems. Building large information systems, and maintaining them over long periods of time, has been shown to be a knowledge-intensive activity [2]. Engineering an information system requires many design decisions. They involve knowledge about functional and non-functional requirements, about conceptual, architectural and physical designs, about implementation languages and strategies, and most importantly, about the relationships between all these levels of knowledge. Recording the knowledge used for decisions—especially important for maintenance and reusability—requires the construction and management of a large knowledge base, and can thus be legitimately viewed as a special case of the knowledge engineering idea. Starting with early work on languages such as TAXIS [3] and RML [4], specialized languages, methodologies and tools for information systems development and maintenance have evolved from this "IS knowledge engineering" paradigm. Of course, these languages, methods and tools must be firmly grounded in results gained earlier in areas of data engineering and software engineering research such as semantic data models, data model mappings, view integration, relational design theory, automatic programming, formally verified refinement, etc.

In this paper, we analyze the data modelling (or—here synonymously—knowledge representation) requirements of such a paradigm and propose a software process data model, together with an associated knowledge base management system, to deal with these requirements. The proposed data model can be viewed as a substantial extension of

an entity–relationship approach which emphasizes process orientation, design decision support and integration of heterogeneous active objects into the software process knowledge base.

There have been a number of efforts to deal with the data management problems of large-scale development and maintenance environments. In the software engineering area, the most popular tools have been enhanced file systems which address the problems of version and configuration control [5]. Traditional database systems have proven less suitable [6] but several projects have extended their concepts by complex objects, versions, redundant derived data (such as compiled programs) and the like [7]. However, there still seem to be several shortcomings of these systems:

- They typically deal with *documents* rather than with conceptual design objects.
- Many of them consider dependencies among documents as a development history. Hardly any systems document the *design decisions* underlying these dependencies or the *tools* used to create them; this, however, is important knowledge for maintenance and reusability. Even fewer control the choice among applicable decisions or tools by enforcing organizational or project *methodologies*.
- Software databases are typically not concerned with *tool integration* and *project management* issues although these are important with long-term software processes.

A more comprehensive approach should therefore stress the process aspect of software development, and must provide more flexibility. Knowledge representation languages which have already been shown to be useful for requirements modelling purposes [4], appear as a good starting point. In essence, software development is seen here as a knowledge engineering process to be supported by a knowledge base management system (KBMS) [8].

Maintenance and reusability are considered crucial knowledge engineering tasks in long-lived information systems. In the context of ESPRIT project DAIDA [9], we have been developing a KBMS called *ConceptBase* which provides a semantic theory of objects, processes and tools in a heterogeneous information systems development and usage environment, together with the computational facilities of a software database. Together with a semantic theory of the application domain and of the system requirements (expressed in the same knowledge representation language), such a KBMS is intended to control and document a historical account of:

—*what* the information system knows about the world,
—*how* the information system fits into the world,
—how and *why* these two kinds of system requirements were mapped into the design and implementation of an information system.

We wish to maintain this information to facilitate maintenance and reusability of software objects not only at the code level, but also at the levels of user requirements or conceptual designs. Indeed, we intend to reuse design *process experiences* rather than just their outcomes.

The model described in this paper represents a first step towards such a goal. Formally, it can be viewed as an extension of the entity–relationship model in databases [10], of Petri net structures [11], or of incremental and iterative design methods proposed in AI and software engineering [12, 13]. Specifically, the main ideas are:

- To represent the evolution of design *objects* by *tool*-aided design *decisions*:
  —covering conceptual design *objects* as well as software documents,
  —viewing design *decisions* as special kinds of design objects that are explicitly represented, can be justified by other decisions, and may evolve over time,
  —viewing design *tools* as reusable design decisions, intended to support the execution of other design decisions;
- To exploit the instantiation hierarchy of an *extensible knowledge representation language* for integrating heterogeneous languages, methodologies and tools:
  —defining the process model at the metametaclass level,
  —defining a particular software development environment at the metaclass level,
  —documenting a particular software development project at the class level,
  —prototyping a particular design at the instance level,†
- To integrate design-in-the-large with design-in-the-small issues:
  —offering modularization of the knowledge base, in particular of design decisions, while providing semantic descriptions at all levels,
  —allowing flexible precision of software process control, potentially ranging from pure database functionality (no semantic description) to rather detailed temporal and/or predicative assertions.

After a brief overview of the DAIDA project as a whole (which also relates our work to that of others), Section 2 studies detailed requirements for a decision-centered approach to conceptual software process modelling. Section 3 briefly reviews the conceptual modelling language CML, viewed in our system as a

†In this paper, we shall not discuss prototyping further although it is part of the DAIDA project. Therefore, we usually simplify the model so that the process model is described at the metalevel, an environment at the class level, and a software project at the instance level.

hybrid knowledge representation mechanism which integrates semantic networks, rule-based systems and frames. Section 4 then applies this language to formalize the software process model, using the same example as in Section 2. Section 5 briefly describes the ConceptBase prototype implementation. Finally, Section 6 presents several applications in the DAIDA context, in particular the representation of mapping requirements to design and design to implementation, as well as use of the process model in the ConceptBase implementation itself.

2. REQUIREMENTS OF A DECISION-BASED SOFTWARE PROCESS MODEL

This section is devoted to analyzing the *requirements* for a KBMS that supports an environment for information system evolution. First, we characterize the concrete context in which we are working, i.e. the DAIDA system. Then, we give a simple development and maintenance example to provide an intuitive feeling of what kind of support is needed. Finally, we outline and justify requirements for a conceptual model which relates the design objects and documents generated in a software environment to the tools used to generate them by a notion of *design decision*. It is sketched how the combination of this decision-centred approach with object-oriented construction principles may address a large number of problems arising in database software evolution.

2.1. DAIDA project objectives

It is the goal of DAIDA to exploit some specific properties of data-intensive information systems to come up with a specialized design KBMS which can take maximum advantage of this application knowledge. A decision-based documentation methodology is chosen to support consistent maintenance, reusability and configuration of multi-layered descriptions. The architecture, summarized in Fig. 1, is based on the following concepts and observations:

1. **Multiple levels of representation**—DAIDA views an information system as a multi-layered description of requirements analyses, designs and implementations [8]. The layers are represented in similar but distinct languages: the knowledge representation language CML/Telos [14, 15] for requirements analysis; a purely declarative version of the language Taxis [3], called TaxisDL [16], for conceptual design and predicative specification; and the database programming language DBPL [17] for implementation design and programming. Note that there is a break in paradigm in the middle: CML and TaxisDL are object-oriented conceptual models of the world, and of the system embedded in it, but have to be transformed into a set-theoretically motivated database programming language.
2. **Extensible set of interrelated transformation assistants**—The literature has developed a rich set of transformation rules for refining and implementing specifications. For example, the CIP [18] and REFINE [19] projects propose user-guided formal transformation strategies, whereas the Programmer's Apprentice [20] views a program as a puzzle of adaptable clichés which must be maintained in a consistent state in case of changes, using dependency-directed

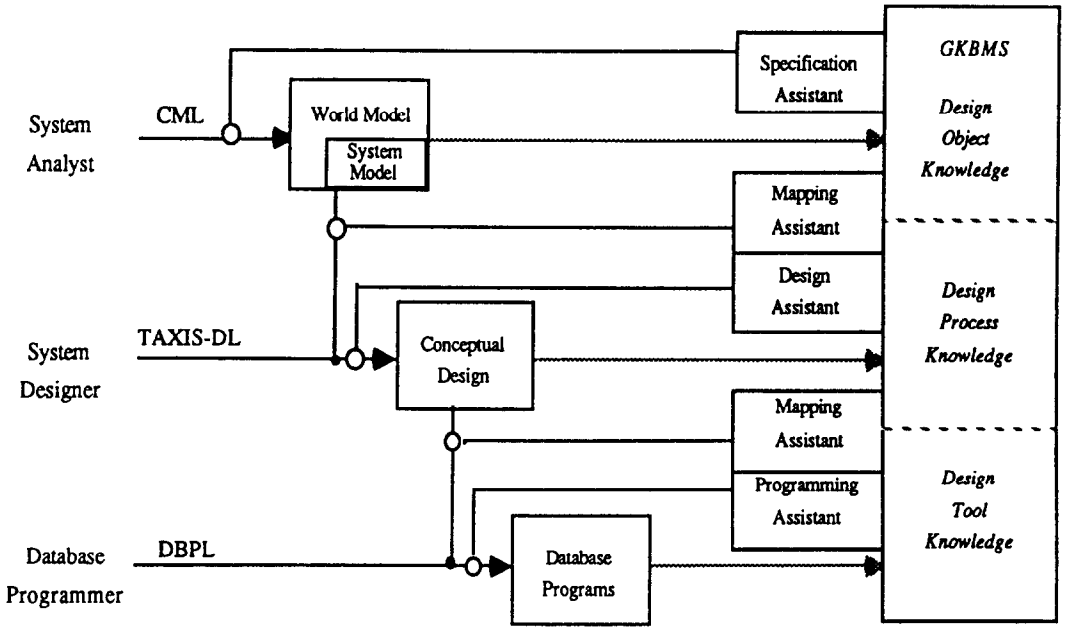

Fig. 1. DAIDA architecture.

backtracking strategies. Most of these tools have been successful only for programming-in-the-small, whereas information systems are often quite large. Therefore, DAIDA provides a flexible "open" environment which can support a range of development situations from (almost) manual to (almost) automatic, depending on the currently available set of transformation tools. To achieve this, transformation tools are embedded in a fairly large number of small "expert systems", called assistants, which communicate via the common knowledge base to be described below; due to the multi-layered structure of DAIDA, *language assistants* for each level must interact with *mapping assistants* between the levels. The application domain of DAIDA, data-intensive information systems, cannot only exploit general software development expertise, but also the special representations, theoretical results and methods of database design research. Moreover, certain mathematical transformation methods, as e.g. expressible in Z [21], appear particularly suited for this application domain. Specifically, the need for assistants in three major transformational tasks results from the above-mentioned levels of languages:

- *embedding* a CML system model in the CML world model, and *narrowing* it to a TaxisDL conceptual design, remaining in the object-oriented framework [22],
- *validating* the CML and TaxisDL models by *prototyping* (in DAIDA, this is done in an object-oriented extension of Prolog [23]),
- *refining* the object-oriented specifications towards set-theoretic database programming, using Abrial's set-theoretic substitution calculus and B-tool [24].

3. **Formalization of information systems requirements**—Most formal software development methodologies start with a formal specification of system functionality. Formalizing the requirements analysis which leads to these specifications, has been traditionally considered difficult or even impossible. Again, the concentration on data-intensive information systems improves the situation. Database schemata naturally represent a system model of the relevant world domain; the analysis underlying the development of the initial database schema can be reused as a starting point for the requirements analysis of new applications. However, a knowledge representation language more powerful than traditional data definition languages, even for semantic data models, is required to describe the relationship of the system model (as in the database schema) to the world model, and the development of this relationship over time. The conceptual modelling language CML [14, 15], evolved from the requirements modelling language RML [4], offers an object-oriented model with an embedded time component to support this task.
4. **Integrated decision-based documentation knowledge base**—Representing multiple layers of system description as well as their relationship to a description of the underlying real world can offer powerful development and maintenance support for information systems but requires itself a knowledge base management system for maintaining the different descriptions consistent over time: the DAIDA global KBMS (GKBMS). Rather than just modelling (versions of) development *objects*, the GKBMS views the software development and maintenance process as a *history of tool-supported decision executions*. These decision executions are directly represented, they can be planned for, reasoned about and selectively backtracked in case of errors or requirements changes. *Ex ante*, the GKBMS can be seen as an integrative tool server which helps users in selecting tasks and tools within a large development project; *ex post*, it plays the role of a documentation service in which development objects are related to the decisions and tools that created or changed them (i.e. justify their current status). Many recent ideas from design database research [25] apply to the implementation of such a system; applying the DAIDA philosophy to the GKBMS (viewed as a data-intensive information system about the history of "software worlds"), a dialect of CML is chosen as the knowledge representation language. ConceptBase is a prototype system that implements both CML itself and the GKBMS model on top of it.

2.2. A DAIDA example

Based on the architecture in Fig. 1, Fig. 2 illustrates a simple DAIDA development process, using the example of an information system for project meeting support [26]. A CML *world model* starts from the activity, *Meeting*, within a project and describes its related activities and entities in a real world with time. Among other things, meeting preparation, conduction and follow-up is different for people in different roles, namely organizers and other participants. Based on this observation, the CML *system model* is positioned in the world model in two functional parts (also called system activities or views), one supporting an organizer, the other a participant within the same, given organization.

The combined world and system models are mapped to a TaxisDL *design model*. The role of the system model within long-term world model activities is represented by a script, *office-internal meeting schedule*; certain aspects of other activities and data are mapped to data classes, transaction classes and their corresponding constraints. Within the TaxisDL

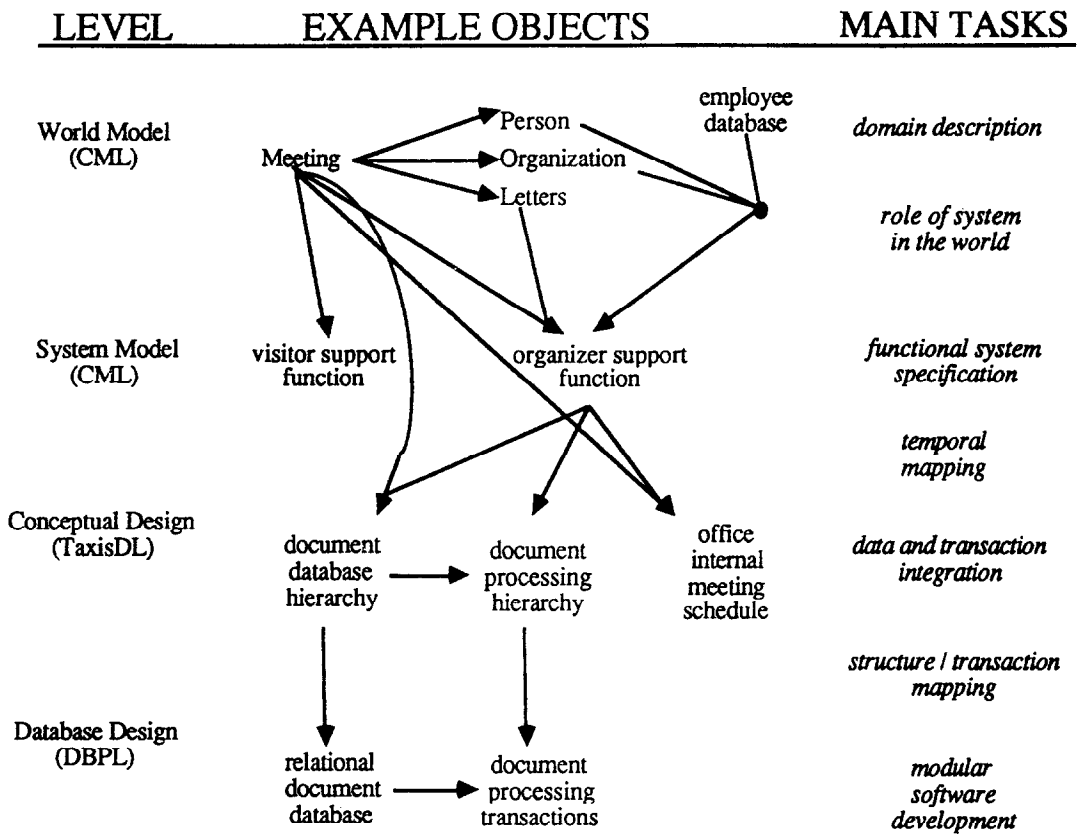

Fig. 2. Overview of the development example.

model, data class hierarchies and corresponding transaction hierarchies must be synthesized from the mapping results, to achieve an integrated conceptual design; this could be called a particular strategy for *view integration*, to be supported by the TaxisDL knowledge-based design assistant. In our example, we detected that from the various outputs of meeting we could compose a conceptual office document data-

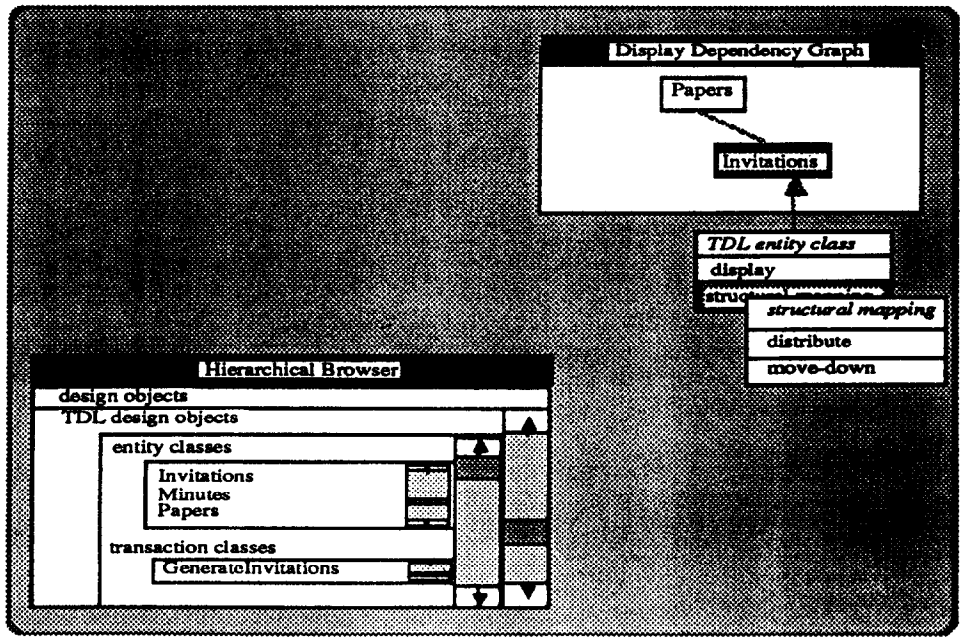

Fig. 3. Browsing design objects on an *IsA* hierarchy of the conceptual design.

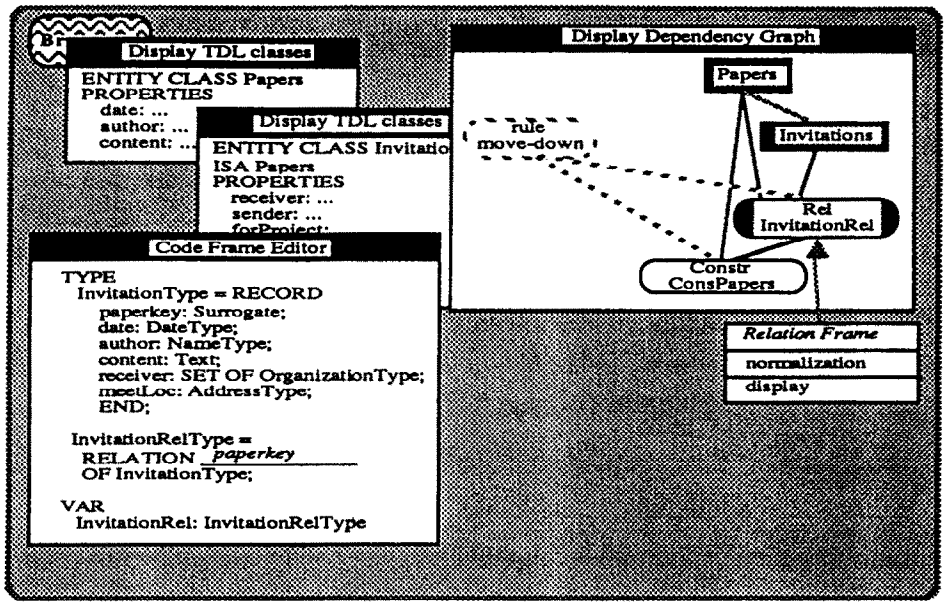

Fig. 4. Graphical display of dependencies and code frames generated by mapping rules.

base, consisting of expense notes, working papers, invitation letters, minutes and the like.

The design is mapped to a DBPL *database structure and transaction design*. Decisions involved in mapping the TaxisDL generalization hierarchy of papers and the related transaction hierarchy to a modular DBPL program with relations, views, integrity constraints and database transactions [24], are presented below in a highly simplified manner to elicit GKBMS requirements.

In Fig. 3 (screens simplified for readability in this section), the developer has employed a *hierarchical text browser* to determine unmapped TaxisDL objects. He has further decided to *focus* on the mapping of entity structures, in particular, invitations and their generalization, papers. This selection causes the display of a *menu* with *applicable decision classes and tools*. There are several possible mapping strategies [27, 28]; *distribute* would generate one relation per TaxisDL entity class, whereas *move-down* only generates relations for leaves of the hierarchy and represents the other ones by view definitions (called *constructors* in DBPL [29]).

The graph in Fig. 4 shows *dependencies* created by the decision for *move-down*, relating the new objects to existing ones and to a representation of the applied tool. Then, selection of the node *InvitationRel* causes display of the corresponding sources (type and variable definitions).

Invitation Type contains a set-valued attribute; a *normalization decision* is therefore offered in the menu, leading to the extended dependency graph in Fig. 5. The new *selector* expresses the referential integrity constraint among the two relations, whereas the new *constructor* allows the reconstruction of the initial, unnormalized invitation relation; for details, see [26]. Additionally, Fig. 5 demonstrates how *automatic and manual execution of decisions* could interact. Observing that the system contains only *Invitations* and no other *Papers*, the developer decides to "make the system more user-friendly" by replacing the artificial *paperkey* attribute (initially required to map the object-oriented TaxisDL model which does not have keys) with *date, author*. Of course, this change also implies adaptation of the corresponding constructor, selector and possibly transaction definitions (outside the editor window in Figs 5 and 6).

Unfortunately, the assumption that *Invitations* are the only kind of *Papers* leads to an inconsistency as soon as the mapping of *Minutes*, the second subclass of *Papers*, is considered (Fig. 6). Therefore, the decision to choose associative keys must be *retracted*, together with all its consequent changes, without redoing all the rest of the design; supporting this consistent, selective backtracking is one main purpose of introducing the explicit documentation of design decisions and dependencies. In the example, the inconsistency can be resolved by selectively backtracking to the state before the introduction of associative keys; in other cases, or if the granularity of representation in the dependency graph is insufficient, additional manual or tool-aided corrections may become necessary. Note that the graph in Fig. 6 only highlights the objects to be changed when introducing *Minutes*; the actual correction would need a more detailed representation—the GKBMS must have some kind of *zooming* facility for both design objects and design decisions.

2.3. Requirements for a process-oriented software information system

Although the above example is highly simplified compared with real-world software projects, a number of requirements for effective KBMS support

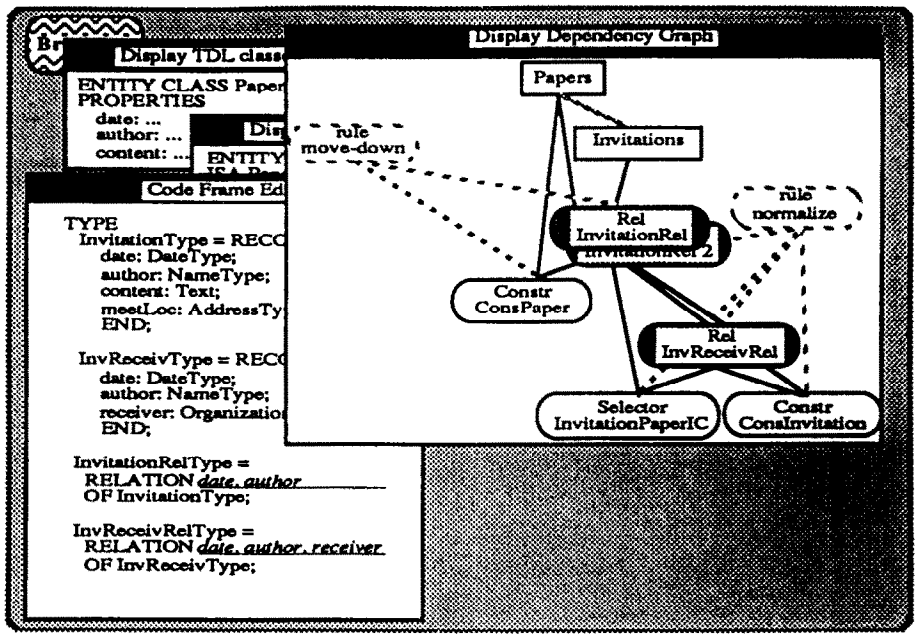

Fig. 5. Dependency graph and code frames after normalization and key substitution.

should have become obvious. First, we have a need for representing design *objects* or documents at different levels of abstraction, and at any of the DAIDA language layers. Second, the GKBMS must know about *tools* for supporting intra-language refinement (e.g. normalization within DBPL) and inter-language mapping (e.g. generalization hierarchy mapping). Third, a *usage environment* must offer interface tools, including object *and* task dependent menus, and the documentation of design object interrelationships, both embedded in some methodology to aid in the *process* of software development and, especially, software maintenance (e.g. retraction of user-defined keys in Fig. 6).

In fact, process support is the central concern of our approach. In our view, the software process is based on human *design decisions*. When executed, these decisions lead to certain transformational oper-

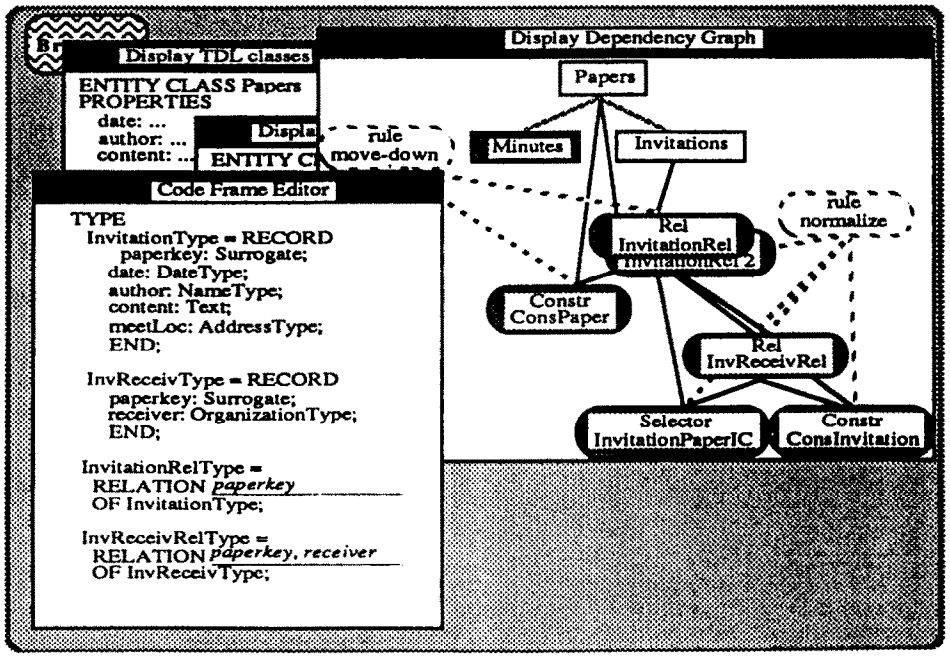

Fig. 6. Code frames and dependency graph after backtracking the decision on key substitution.

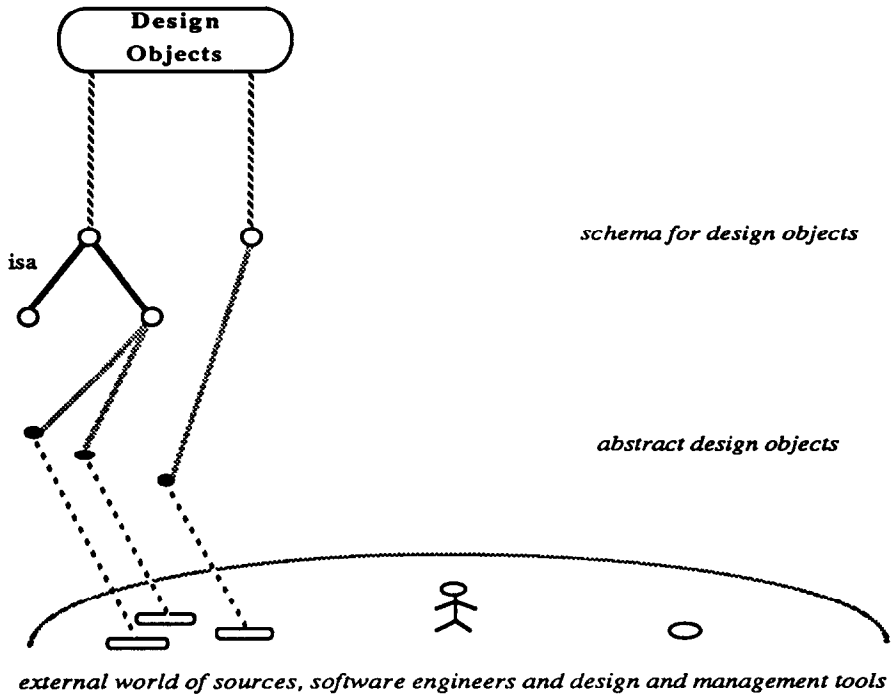

Fig. 7. Design object knowledge base structure.

ations in the software environment; transformations establish relationships between design objects and may be supported by tools. However, in a large software project, software developers may not be allowed to select arbitrary tools from, say, a toolkit [30], to work arbitrarily on arbitrary objects. Rather, a *methodology* with associated standards should be enforced, constraining working sequences and tool applications in a meaningful, theory-based manner, as far as possible without impeding developer creativity. To allow the KBMS such a flexible definition of methodology which could range from very open to very formal, we introduce the notion of *decision class* of which any design decision execution must be an instance. Thus, we propose to couple object-oriented construction principles with the notion of design decision; in contrast to usual object-oriented systems like SmallTalk [31], tools (called methods therein) are *not* directly associated with object classes but only indirectly via decision classes. In the following, the requirements for the approach sketched above will be outlined in more detail.

Although our main focus is the representation of software processes, it appears best to start with discussing the representational requirements for design objects. The term ***design object*** denotes any software object and document involved in world or system modelling, system design or database programming. Note that in a heterogeneous software environment like DAIDA, design objects reside *outside* the GKBMS and are represented in languages *not understandable* for the GKBMS.† To deal with external and unintelligible design object sources, simple configuration managers [30] just represent *source references*. This prevents any deeper reasoning about design object semantics and interrelationships with other design objects, decisions and tools. Taking a *knowledge* management view, design objects should not only have a source reference but also formalized knowledge *about* the sources, and of the design decisions that influenced their evolution. The control of such a representation requires at least five levels of abstraction (Fig. 7):

(a) management of specific design object *sources* (software documents), often residing in a file system such as UNIX under simple configuration control;
(b) *knowledge about specific design object instances*, to document the sources in a formal way and to reason about their interrelationship (e.g. configurations, versions);
(c) *knowledge about design object classes* to gain a powerful structuring mechanism which defines the possible objects appearing in a particular software environment (e.g. world model, system model, TaxisDL and DBPL constructs in DAIDA);

†However, there is at least a possibility to activate and control these external design objects (e.g. DBPL programs) and their building environments automatically. This is in contrast to CAD applications relating to non-computer projects [32], but similar to CIM applications where the developed designs control and activate flexible manufacturing equipment.

(d) *a system-understandable terminology* to talk about design objects, defining formally the GKBMS approach to modelling software objects;
(e) *a knowledge representation language* to realize all of the levels above.

This five-level model can be used to characterize the flexibility of software databases (e.g. [33,6]). In particular, the knowledge representation language defines how precisely knowledge about objects can be described, and how easily the object schema at level (d) can be adapted to other languages and tools. Since new languages, methods, theories and tools for software development are continuously appearing, *extensibility* of the language as well as of the object schema is of great importance; it is well-known that this implies the use of generalization (*IsA*) hierarchies of object classes [34–36]. We experience the need for extensibility in the DAIDA project where languages and tools evolve rapidly, as our research progresses.

Despite the large amount of knowledge that can be made available in such a schema, design object representation really only covers the static aspects, i.e. the *outcomes* of development processes. Therefore, we introduce conceptual models of ***design decisions*** as first-class objects intended to control and document directly the development *process* that creates, alters and justifies design objects. As indicated before, design decisions play multiple roles in our approach and must be adaptable to multiple levels of granularity (ranging from programming-in-the-small to programming-in-the-large to programming-in-the-many [30]) as well as to multiple methodologies. A single set of evolution rules for a predefined object schema, as given e.g. in [37], is very useful in a well-understood task but not enough for a heterogeneous environment; moreover, we want to preserve human discretion in making decisions about software evolution, rather than prescribing rigid rules. As a consequence, the same five-level representational requirements as for design objects apply to the modelling of design decision knowledge:

(a) *design decisions* made and executed in the external world, possibly collaboratively by (groups of) human designers and computerized problem solvers;
(b) knowledge about *executed design decision instances*, possibly including limited documentation of the decision-making process;
(c) knowledge about *feasible classes of design decisions* according to known development theory, standards or methodologies;
(d) a *terminology and associated enforcement system* for design decisions that formally defines the GKBMS model of design decision control and documentation;
(e) a *knowledge representation language* to represent knowledge at all of the above levels.

The same remarks as before apply with respect to the need for extensibility of language (e) and schema (d). For example, in an evolving software environment such as DAIDA, this extensibility allows developers to use the GKBMS initially as a simple documentation tool where all transformations are made manually, and recorded and controlled according to very simple decision class definitions, basically just distinguishing between three kinds of decisions: refinement within a language, mapping between languages and retraction of existing decisions to start new versions. This distinction is closely related to a versioning model described in [32], and can thus serve as a basis for certain programming-in-the-large tasks. As theory and tools for the mapping tasks sketched in Section 2.2. are further developed, the same schema can support an almost automated software development and maintenance process.

Finally, ***design tools*** employed to execute decisions can be described in a fashion similar to design decisions, namely, at a class level which describes what the tools can guarantee to do in general, and at an instance level which describes what it guarantees in executing a specific decision. The role of tool modelling is best understood by studying the *interrelationships between design objects, design decisions and tools*. Figure 8 extends Fig. 7 to illustrate these interrelationships. For example, at the class level, a design decision class should be related to object classes and tool specifications as follows:

- Design object classes this decision can be applied to (*FROM*)
- Design object classes allowed as outcomes achieved by performing this decision (*TO*)
- Associated tools supporting the execution of a decision (*BY*)
- A formalized description decomposing a decision in subdecisions, and finally into primitive dependencies among incoming and outgoing design objects
- A decision-procedure description (maybe just a kind of comment) capturing developers' beliefs not expressible in the above representation.

Furthermore, both decision class and tool specifications come with constraints that define the relationships between inputs and outputs. For decision classes, the semantics of such a constraint is similar to that of an integrity constraint in a database transaction [38]: the constraint must be satisfied for any completed instance of the class. For tool specifications, the semantics of a constraint is that of a warranty the tool gives to its users; in particular, satisfaction of constraints already guaranteed by the supporting tool need not be checked any more in the instantiation of a decision class (so to speak, at transaction end), unless there was a chance for the user to invalidate the tool results in between. The implementation of such an approach requires a theorem-proving approach to integrity checking [39].

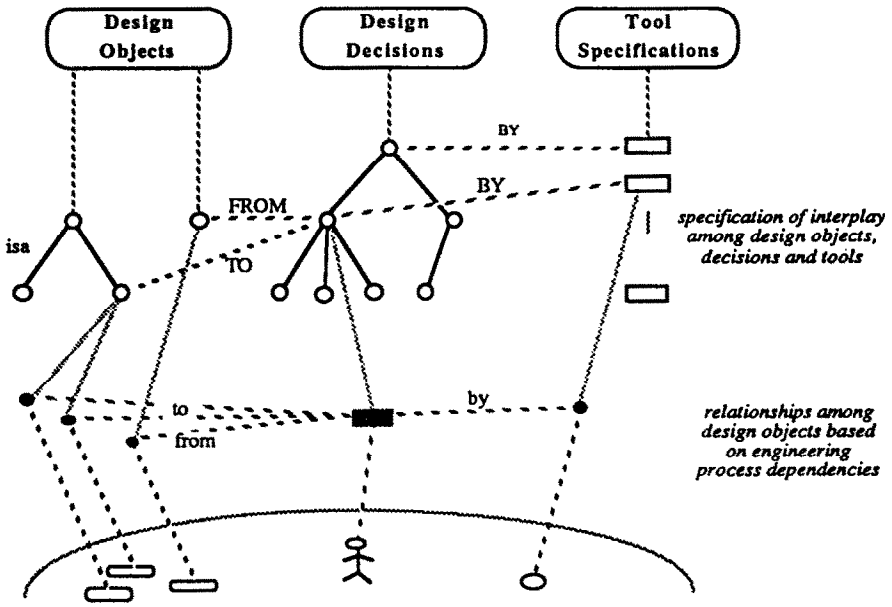

Fig. 8. Combining the design decision concept with layered knowledge representation.

For simplicity, the *decomposition* and *decision-procedure* components of the model are not shown in the figure; the former will be discussed when elaborating the formal model of design decisions in Section 4.3, whereas we have only begun to explore the latter. Another important requirement is the modelling of *time*, an important aspect of any process-oriented model. We argue that an *interval-based* model of time [40] should be chosen since it models aspects such as versioning of design objects, or embedding of validity intervals for design decisions—as implied by the decision decomposition approach mentioned above. Finally, it may be useful to add another level of abstraction to the model, in order to represent example data for *prototyping* in the model; this would make the levels (b–d) above into classes, metaclasses and metametaclasses, respectively. Since we do not discuss prototyping further in this paper, we shall stick with the simpler form although ConceptBase supports this extension as well.

So far, we have focussed on *representational* requirements for a decision-oriented GKBMS. In order to get a feeling for the ***functional requirements***, we now discuss how a typical mapping task such as illustrated in Section 2.2. could be supported by the structure shown in Fig. 8. First of all, different *exploration* facilities are required to exploit the documentation of design object and design decision representation during the development and maintenance phases:

- Exploration of hierarchical structures such as taxonomies of design object or design decisions classes, possibly also of documented instances and their static relationships, starting from a given focus; e.g. input/output relationships between DBPL transactions and data structures (*browsing of outcomes*)
- Exploration of dependency graph structures, following chains of design decision instances at various levels of granularity from a given focus; e.g. finding requirements and design decisions a relation attribute was derived from (*browsing of processes*)
- *Predicative restriction* of a set of design objects and design decisions (e.g. for *setting a focus* or for reducing the complexity and size of a display)
- Combined navigation in graphs starting at a given focus; e.g. explore the design object space at the level of system design, then explore possible implementation decisions.

From this list, it is obvious that a combined predicative and direct-manipulation style of interaction is needed for the KBMS usage environment. Exploration of the existing schema and instances is required both during the initial development of a system and in the maintenance phase. In a typical development step, the interplay of design objects, decisions and tools could proceed as follows:

1. Explore (versions of) design objects and decisions (*instance level*).
2. Select a design object to work on (*instance level*) and finds its class (*class level*).
3. Explore decision classes applicable to this object class and select one (*class level*).
4. Select a tool associated with the selected decision class or one of its predecessors in the generalization hierarchy of decision classes (*class level*).
5. Make a decision within this class, execute it with

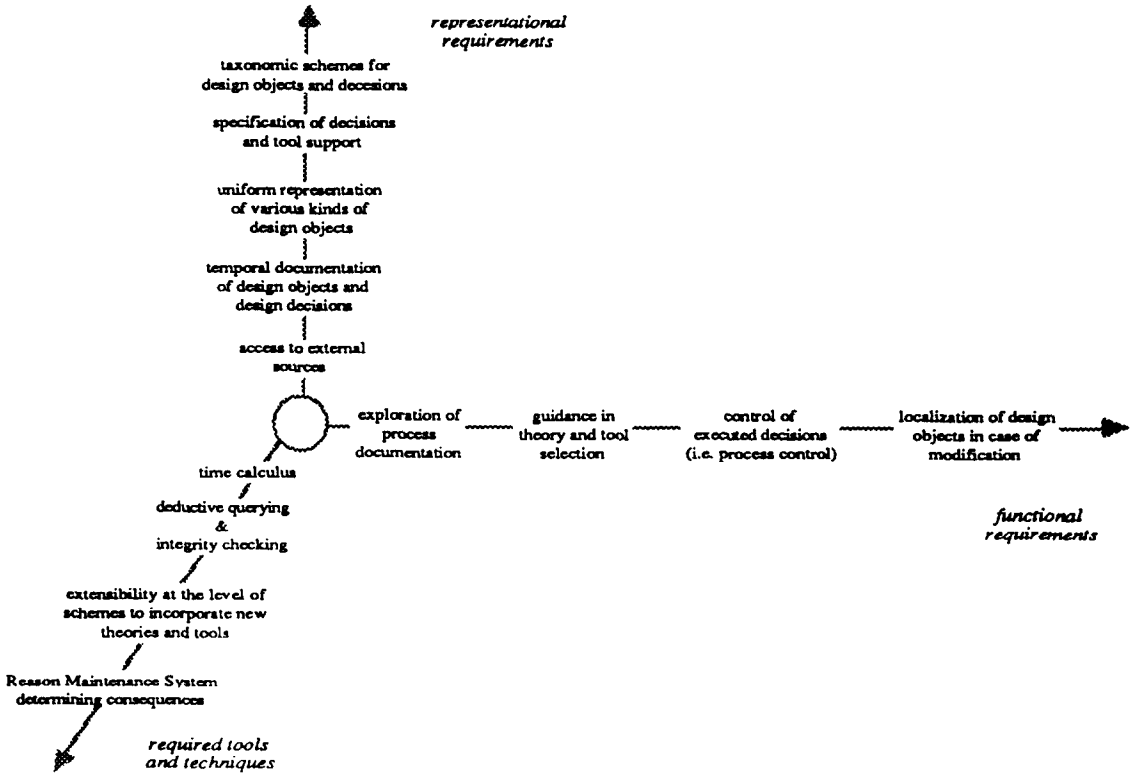

Fig. 9. Summary of KBMS requirements for software process support.

the selected tool, generating new design object sources (*external world level*) and their representations in the knowledge base (*instance level*), testing if these instantiate existing design object classes (*class level*).

6. Try to create an instance for the previously chosen decision class, testing the correctness of the execution with respect to the class definition and, if successful, documenting the execution with its associated objects and tools (*instance and class level*).

Introducing design decisions as a mediating concept between objects and tools *guides* the user towards applicable tools in a given task context (defined by the theory or methodology embedded in a decision class definition), *controls* the correct application of these tools in a flexible way (using weaker or stronger constraints for decision classes) and *documents* the development process for subsequent *explanation*, critique (*maintenance*) and *reuse*. In the long range, it would be desirable if the system would extend its known set of decision classes by inducing new subclasses from instances [2, 41, 42].

Summarizing, ***three dimensions of requirements*** for modelling and supporting software processes in a knowledge base have been pointed out:

—*representational requirements* for a software process data model (GKBMS data model)
—*functional requirements* (operational interface of the GKBMS)
—*required tools and techniques* (implementation of the GKBMS).

The details of these dimensions are repeated in Fig. 9. In the remainder of this paper, we present our approach to satisfy these requirements. The *knowledge representation language* mentioned at level (e) above for modelling both design objects and design decisions must combine object-oriented abstraction with multiple levels of instantiation, one or more assertion languages for expressing object and process constraints, natural concept visualization with predicative as well as navigational exploration, an embedded (preferably interval-based) model of time, and object identity as a basis for configuration management. Taken together, these requirements look very similar to those needed for world and system modelling in DAIDA; indeed, a software environment can be seen as a "software world" whose structures, laws and history have to be represented in the GKBMS. As a consequence, we choose a dialect of CML, the world and system modelling language of DAIDA (cf. Section 2.1), as the knowledge representation language for the GKBMS.

The next section presents a definition of this CML dialect. Then, the level (b–d) representational requirements are addressed by defining formal constructs for design objects and design decisions. Continuing the example of Section 2.2, our approach to the functional requirements is also briefly demonstrated. Finally, we present the *tools and techniques* aspects by

giving an overview of the *ConceptBase* prototype implementation, and relate the model to specific applications.

3. THE CONCEPTUAL MODELLING LANGUAGE CML/TELOS

This section provides a brief review of the knowledge representation language CML which will serve as the basis for formalizing and implementing our software process knowledge base. CML (and its minor variants SML and Telos [15]) was derived in several iterations [14, 43] from the requirements modelling language RML [4], and has been augmented in DAIDA with special features for modelling system requirements and external naming for system-generated object identifiers.

CML combines structurally object-oriented principles such as object identity, classification, generalization and aggregation, with a predicative assertion language and a built-in time calculus. Major features distinguishing CML from other similar knowledge representation languages include:

- attributes as first-class objects which can be instantiated, specialized and have attributes of their own;
- potentially infinite hierarchy of metaclass levels, thus ensuring extensibility of the language;
- validity intervals for world objects described in the system, as well as for the system's knowledge about them;
- flexible hypertext-like syntax that allows for arbitrary combination of semantic network and frame-based views.

The remainder of this section sketches the network (proposition) and the frame (object) levels of the system as well as their interrelationships. A knowledge-level formalization of the basic language can be found in [43].

3.1. The network syntax

In CML, knowledge bases are seen as *semantic networks*. A link (which is synonym to object in CML) is interpreted as the proposition stating that there is a connection between two nodes. A node represents the proposition that there is such an object. The object-oriented paradigms of **classification, generalization** and **aggregation** [34] appear as links, too, where a set of six language axioms defines the well-formedness of the network. For example, each object has to be an instance of at least one object (its class). The uniform data-structure for propositions is:

$$\text{id} = \langle \text{source, label, destination, interval} \rangle.$$

Each proposition makes a statement about objects and is itself an object. On the left stands the name (id) of the statement, and on the right the definition: the object "source" has a link labelled "label" to object "destination" during time "interval". Nodes are seen as self-referential links, so-called *individuals*, denoted by $id = \langle id,\_,id,interval \rangle$, where the underscore stands for an arbitrary label. Obviously, individuals make no statement about other objects but only about themselves; more exactly, they state that there is an object with name "id."

To support rule-based deduction and integrity control, CML offers specialized object classes to express ***constraints*** and ***rules***. For example, a proposition can link a class object to an object of class "ConstraintClass" to express that the constraint has to be satisfied for all instances of that class object. Note that this method of introducing assertions leaves the freedom to attach arbitrary assertion languages and associated provers to the system [44].

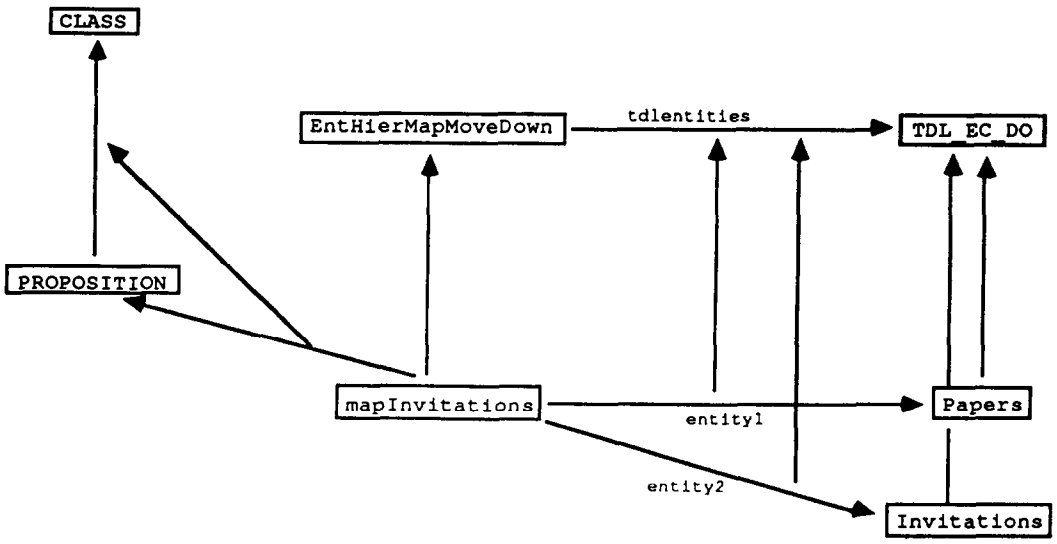

Fig. 10. Propositional representation of mapInvitations (unlabelled links stand for *instance) of propositions.

3.2. The frame syntax

By grouping a set of propositions together with their class propositions around a common source, we obtain a CML frame. For example, a piece of a frame-level object *mapInvitations* that documents the design decision shown in Fig. 4, can be written as:

```
PROPOSITION mapInvitations at version7
  IN EntHierMapMoveDown WITH
  tdlentities
        entity1: Papers
        entity2: Invitations
END (* mapInvitations *)
```

This states that mapInvitations is an instance of the classes *PROPOSITION* and *EntHierMapMoveDown* (the decision class activated in Fig. 3). It has attributes *entity1* and *entity2* with values *Papers* and *Invitations* which are instantiated from an attribute category labelled *tdlentities* (defined in class *EntHierMapMoveDown*). *Version 7* denotes the time during which the frame shall be regarded as valid. Part of the network of propositions representing the frame is shown in Fig. 10.

Figure 10 also illustrates one of the CML axioms. The attribute labelled *entity1* (*entity2*) is declared to be an instance of the *tdlentities* attribute of *EntHierMapMoveDown*. The instantiation axiom of CML demands that its source *mapInvitations* must be an instance of the source of its class *EntHierMapMoveDown*; also, *Papers* (*Invitations*) must be instances of *TDL_EC_DO*.

The time components of the propositions are not shown in the figure; for example:

```
mapInvitations = ⟨mapInvitations, −, mapInvitations, version7⟩
P1 = ⟨mapInvitations, *instanceof, PROPOSITION, version7⟩
P2 = ⟨P1, *instanceof, InstanceOf, 21-Mar-1989+⟩
...
P8 = ⟨mapInvitations, entity2, Invitations, version7⟩
P9 = ⟨P8, *instanceof, K1, version7⟩
P10 = ⟨P9, *instanceof, InstanceOf, 21-Mar-1989+⟩
where
InstanceOf = ⟨PROPOSITION, *instanceof, CLASS, Always⟩
K1 = ⟨EntHierMapMoveDown, tdlentities, TDL_EC_DO, Always⟩
```

The first propostion declares *mapInvitations* as an individual. Its last component, *version7*, holds the "valid time" of the object: the knowledge base regards *mapInvitations* as valid during the time interval *version7*. *P1* instantiates mapInvitations to the class *PROPOSITION*. The next proposition makes *P2* an instance of the class *InstanceOf* (the class of all instantiation links). Its time component is used to store the "belief time" of *mapInvitations* and *P1*: the knowledge base knows of them since *21-Mar-1989*.

CML treats all propositions (individuals, attributes, instantiation and specialization links) as objects. Since many object identifiers like those for attributes and instantiation links are system-generated, we extend the frame syntax by operator expressions that reference links by their source and label components. For example, the identifier *P8* can be referenced by the expression *mapInvitations!entity2*. The operator "!" can be iterated for accessing more distant links: the name of the instantiation link of the entity2 attribute can be described as *mapInvitations!entity2!*instanceof*. At any given point in time, this naming convention yields unique identifiers since the CML aggregation axiom says that there may be only one link with a given label at a given time.

3.3. Querying and updating knowledge bases

Due to the close relationship between the two syntax variations of CML, queries and updates can be addressed to either of them; for simplicity, we assume for the moment that internally, all frame structures are converted to network structures, as indicated in the example above [45]. Following [46], CML views the knowledge base as an abstract data type with two operations:

```
tell (s)
ask (q, a)
```

"tell" tests "s" for consistency with the knowledge base and stores those propositions of "s" not already retrievable. Applied to some knowledge base, "ask" provides the answer "a" to query "q". In accordance with the hypertext-like structure of the language, queries can be asked and answers can be displayed as text (frame) objects, networks or combinations of both. "q" can either be a closed predicative formula over the knowledge base in which case "a" takes one of the values yes, no or unknown; or "q" can be considered a class definition of CML and "a" contains all the objects classified as satisfying this definition (cf. Section 5.1).

The following query asks for all attribute values of all instances of the class *EntHierMapMoveDown*

which are valid during *version7*:

```
INDIVIDUALCLASS AttributeQuery IN QueryClass WITH
    computedattributes
        solution: TDL_EC_DO
    query
        q1: $ each x/EntHierMapMoveDown
            AttrValue (x, tdlentities, solution, version7) $
END
```

Since *mapInvitations* is one of the candidates, the answer is:

```
INDIVIDUAL answer1 IN AttributeQuery/WITH
    solution
        s1: Papers
        s2: Invitations
        ...
END
```

4. FORMALIZATION OF THE SOFTWARE PROCESS MODEL

In this section, the software process model sketched in Section 2.3 will be formalized in terms of the CML language. Recalling the example of Section 2.2, we first formalize the design object hierarchy and then address the modelling of design decisions and methodologies; finally, a discussion of tool specification is provided. In developing this model, especially for design decision control and documentation, we make extensive use of the "!" operator introduced in Section 3.2 to access system-generated attribute identifiers in CML's network syntax. This is shown to yield not only a very compact representation of detailed dependencies among design object properties but also to be directly usable as input to reason maintenance facilities such as [12, 47].

4.1. Overview of the model

As discussed, the software process model represents three basic kinds of objects, namely design objects, design decisions and design tools. The introduction of design tools gives the model an "active database" flavor that distinguishes it from approaches such as entity–relationship [10]. The explicit modelling of design decisions distinguishes it from most previous software databases, and the use of CML's abstraction mechanisms from design process modelling in AI [13]. We first define the metaclasses (actually metametaclasses if prototyping is considered as well) for the three basic kinds of objects (cf.

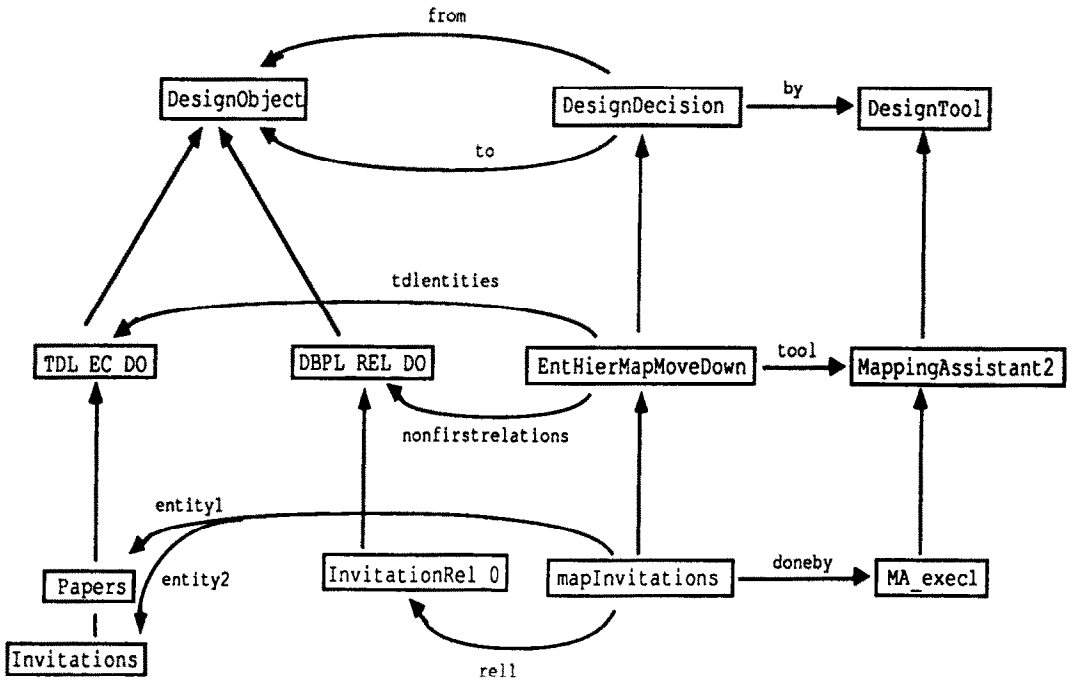

Fig. 11. Overview of the model and example.

also Figure 11); examples of the lower abstraction levels are developed in the remaining subsections.

At the top level, Fig. 11 shows the three metaclasses *DesignObject*, *DesignDecision* and *DesignTool*. Example of design object classes are *TDL_EC_DO* (representing so-called TaxisDL entity classes) and *DBPL_Rel_DO* which can be mapped from the first ones. The tool *MappingAssistant2* helps with such tasks. The lowest level represents actual design objects, decisions and tools. In this case, the mapping of two TaxisDL entity classes to a DBPL relation called *InvitationRel_0* is documented. Note that not all links are included in the figure. The following frame definitions offer a more complete description.

Design objects must be justified by some design decision. Furthermore, the representation of these objects should contain a reference where the source object can be found, as well as a CML description of that object. Finally, a design object may be recursively configured from smaller ones. These requirements are formalized in the CML metaclass:

```
INDIVIDUALCLASS DesignObject
  IN MetaClass WITH
    attribute
        justification: DesignDecision
        objectsource: ExternalReference
        objectsemantic: CLASS
        part: DesignObject
END
```

Instances of *DesignObject* are specialized design object classes corresponding to constructs available in the languages of the chosen environment, in DAIDA CML, TaxisDL and DBPL. In turn, their instances are tokens representing actual design objects defined in one of these languages.

Following the approach of Section 2.3, design objects evolve due to the tool-aided execution of human design decisions under the control of some methodology expressed by decision classes. Design decisions themselves can also be considered as design objects that are worked upon by the design group through other decisions. The CML sub-language for talking about ***design decisions*** is defined by the metaclass:

```
INDIVIDUALCLASS DesignDecision
  IN MetaClass ISA DesignObject WITH
    attribute
        from: DesignObject
        to: DesignObject
        decisionsemantic: DecisionDescription
        by: DesignTool
        part: DesignDecision
END
```

Each instance of *DesignDecision* defines a decision class whose instances in turn record actual decisions. Attribute "from" references the input objects and attribute "to" the resulting objects; time stamps are implicit in the CML language. The "by" attribute refers to the GKBMS representation of the applied design tools. "Part" facilitates the decomposition of design decisions in a modular way. For instance, all specific mapping decisions during a mapping task can be aggregated to a single one covering the whole task.

Our model considers ***design tools*** as design decisions that implement other design decisions classes. The language for talking about tools is defined as a specialization of the metaclass *DesignDecision* where the input to the decision is the design decision class to be supported by the tool, and the output is a procedure that executes the decision:

```
INDIVIDUALCLASS DesignTool IN MetaClass
  ISA DesignDecision WITH
    attribute
        from: DesignDecision
        to: BehaviourObject
END
```

This method of tool integration is intended to consider tools as *reusable software objects* that should, in principle, have been developed with the same methodology as any other software. In the following subsections, the above metaclasses are discussed in more detail.

4.2. Semantic descriptions for design objects

If we wish to know more about a design object than that it exists and where it exists, a semantic description in CML can be given. Note that these descriptions are not equivalent to the sources in the corresponding environments; this is true even for the world and system model (see Fig. 1) where the same language, CML, is used. Nevertheless, the abstract description of design objects in CML helps utilize the structural integrity mechanism of CML for software process control. In the example, we need at least two such classes, TaxisDL entity classes and DBPL relations, for the schema of our software database (containing the objects) respectively knowledge representation (containing object descriptions defined at any CML metalevel):

```
INDIVIDUALCLASS TDL_EC_DO
  IN DesignObject WITH
    justification
        created_by: TDL_Decision
    objectsource
        tdlsource: String
    objectsemantic
        tdlentitydescr: TDL_EntityClass
END
INDIVIDUALCLASS TDL_EntityClass
  IN MetaClass ISA TDL_Dataclass WITH
    attribute
        changing: TDL_DataClass
        unchanging: TDL_DataClass
        unique: TDL_DataClass
        invariant: TDL_DataClass
        setof: TDL_DataClass
END
```

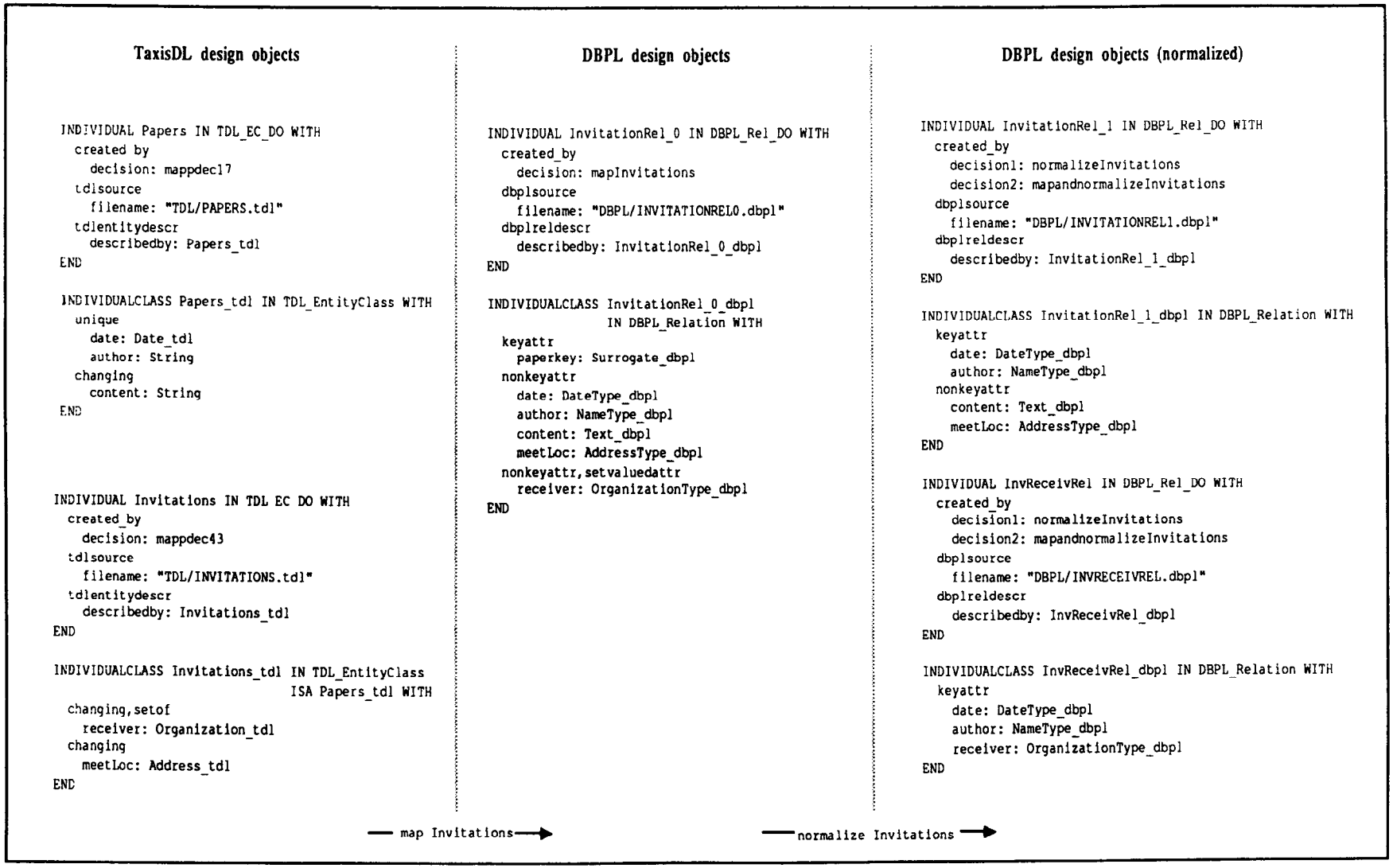

Fig. 12. Evolution of example design objects and their semantic descriptions.

```
INDIVIDUALCLASS DBPL_Rel_DO
  IN DesignObject WITH
    justification
        created_by: DBPL_Decision
    source
        dbplsource: String
    objectsemantic
        dbplreldescr: DBPL_Relation
END

INDIVIDUALCLASS DBPL_Relation
  IN MetaClass WITH
    attribute
        keyattr: DBPL_SimpleType
        nonkeyattr: DBPL_SimpleType
        setvaluedattr: DBPL_SimpleType
END
```

Thus, an instance of *DBPL_Rel_DO* specifies a *DBPL_Decision* for its justification, a filename for pointing to its source file, and a description summarizing the attributes of the relation:

```
INDIVIDUAL Papers IN TDL_EC_DO WITH
    created_by
        decision: mappdec17
    tdlsource
        filename: "TDL/PAPERS.tdl"
    tdlentitydescr
        describedby: Papers_tdl
END

INDIVIDUALCLASS Papers_tdl
  IN TDL_EntityClass WITH
    unique
        date: Date_tdl
        author: String
    changing
        content: String
END
```

This specifies that there is an design object *Papers* justified by *mappdec17* in the TaxisDL environment and that this design object has two unchanging and one changing attributes. Figure 12 completes the design objects of our example. The left side contains the TaxisDL design object *Papers* and its specialization *Invitations*. In the middle, a non-first-normal-form DBPL relation implementing this conceptual design is presented. The two design objects on the right represent the normalized version of *Invitation-Rel_0* used in Figs 5 and 6. They specify for their justification two design decisions which are explained in detail in the next subsection.

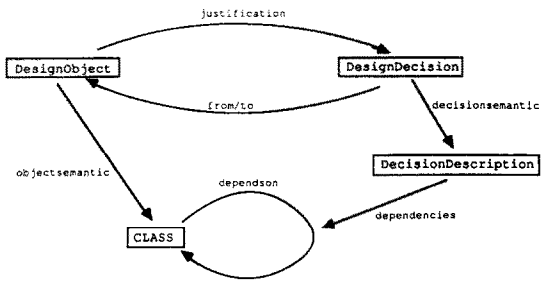

Fig. 13. Metaclass model of semantic descriptions of design objects and design decisions.

4.3. Semantic description of design decisions

The semantics of design decision (at a given level of abstraction) is defined by relating descriptions of design objects to each other. The "decisionsemantic" attribute of metaclass *DesignDecision* is based on special properties of the class "CLASS":

```
INDIVIDUALCLASS CLASS WITH
    attribute
        attribute: CLASS
        dependson: CLASS
END
```

The CML system class *CLASS* defines that classes may have attributes. Above, we extend this definition by so-called dependencies: CML objects may depend on (the existence of) other objects. This can be individuals, attributes, instantiation and specialization relations because they are all objects. When used for the design objects we are able to express how the "object semantic" of the "from" design objects was mapped the object semantic of the "to" design objects.

The class DecisionDescription aggregates such dependencies:

```
INDIVIDUALCLASS DecisionDescription
  IN MetaClass WITH
    attribute
        dependencies: CLASS!dependson
END
```

The semantic network syntax for the extended metaclass model is shown in Fig. 13. Returning to our running example, instances of *DesignDecision* define how to map TaxisDL entity hierarchies to normalized DBPL relations. Recall from Section 2.2. that this requires two steps (or part decisions). The decision class *EntHierMapMoveDown* shows the general knowledge of the GKBMS about how to map a TaxisDL entity hierarchy (like *Papers* and *Invitations* in the previous section) to a DBPL relation which is in general not in first-normal-form. The mapping of the attributes is described by three statements on how the resulting DBPL relation looks:

- the key attributes derive from certain "unchanging" attributes of the TDL entity,
- the non-key attributes are mapped from the other attributes, and
- the set-valued attributes come from the corresponding "setof" attributes.

These constraints are simplified; their full form should include a lot of knowledge about mapping of semantic data models [27] or even complex theories of transaction refinement [24]:

Below, the CML formalization of this class is given. It instantiates the metaclass scheme of design decisions. Formal attributes and dependencies between them are denoted by the "!" operator:

```
INDIVIDUALCLASS EntHierMapMoveDown IN DesignDecision WITH
    from
        tdlentities: TDL_EC_DO
    to
        nonfirstrelations: DBPL_Rel_DO
    decisionsemantic
        mappingdescription: EntHierMapMoveDownDescription
    by
        tool: MappingAssistant2
END

INDIVIDUALCLASS EntHiermapMoveDownDescr IN DecisionDescription
WITH
    dependencies
        keydep: DBPL_Relation!keyattr!dependson
        nonkeydep: DBPL_Relation!nonkeyattr!dependson
        nonfirstdep: DBPL_Relation!setvaluedattr!dependson
END

ATTRIBUTECLASS DBPL_Relation!keyattr WITH
    dependson
        dependson: TDL_EntityClass!unchanging
END

ATTRIBUTECLASS DBPL_Relation!nonkeyattr WITH ... END

ATTRIBUTECLASS DBPL_Relation!setvaluedattr WITH ... END
```

For a visualization of this formalization and its internal compactness, Fig. 14 shows the corresponding semantic network representation. On the left side, the scheme of the software database is defined by the design object and design decision classes. The right side shows how detailed knowledge about software evolution is represented. The design decision *map-Invitations* is an instance of the class *EntHierMap-MoveDown*. It records the actual mapping of the two TaxisDL entities *Papers* and *Invitations* to the unnormalized DBPL relation *InvitationRel_0*. The corresponding instance of *EntHierMapMoveDownDescr*

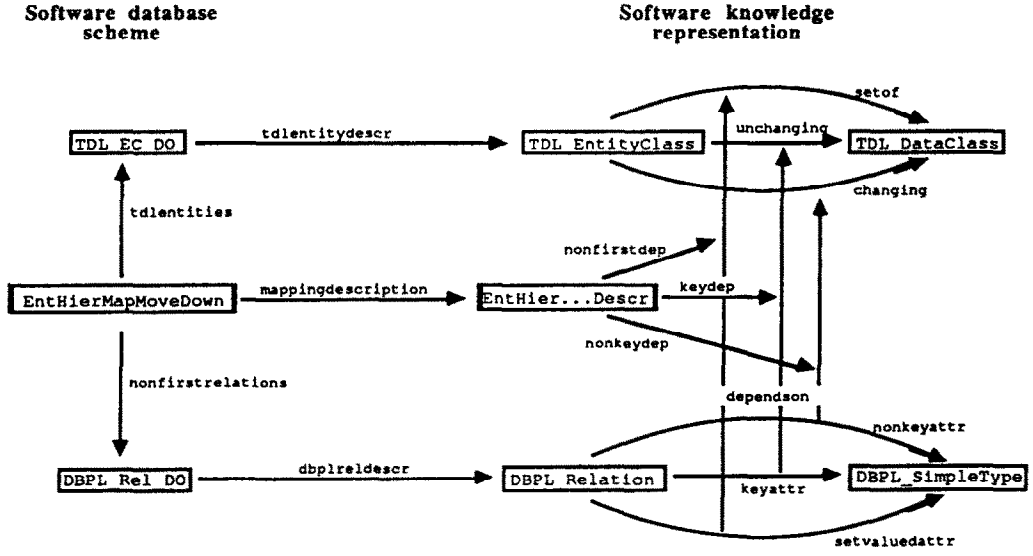

Fig. 14. Design decision class and related object classes with their descriptions.

aggregates the dependencies:

```
INDIVIDUAL mapInvitations IN EntHierMapMoveDown WITH
    tdlentities
        entity1: Papers
        entity2: Invitations
    nonfirstrelations
        rell: InvitationRel_0
    mappingdescription
        describedby: mapInvitationsDescr
    tool
        doneby: MA_execl
END

INDIVIDUAL mapInvitationDescr IN EntHierMapMoveDownDescription WITH
    nonfirstdep
        dep1: InvitationRel_0_dbpl!receiver!depon
    nonkeydep
        dep2: InvitationRel_0_dbpl!meetLoc!depon
        ...
        dep5: InvitationRel_0_dbpl!date!depon
END

ATTRIBUTE InvitationRel_0_dbpl!receiver IN DBPL_Relation!nonkeyattr
WITH
    dependson
        depon: Invitations_tdl!receiver
END

... {same for other attributes}

ATTRIBUTE InvitationRel_0_dbpl!date IN DBPL_Relation!nonkeyattr WITH
    dependson
        depon: Papers_tdl!date
END
```

Figure 15 shows the design object tokens *Papers*, *InvitationRel_0*. The description of *mapInvitations* contains the dependencies between attributes of the *InvitationRel_0*. The description of *mapInvitations* contains the dependencies between attributes of the participating design objects which must be instances of the model shown in Fig. 13; following chains of such dependencies determine repercussions of design modifications, as discussed in Section 2.2.

4.4. Decision modules and methodologies

To summarize the discussion so far, each design decision is characterized by its inputs, outputs and a semantic description, as well as by a pragmatic (tool) characterization of the detailed input–output relationships. While this may be sufficient for small examples and uniform-language situations, it is not enough for large-scale, multi-layered information systems development and maintenance. For this kind of problem, we need a mechanism to aggregate minor decisions to larger ones, or, conversely, to decompose complex decision problems into smaller ones.

The traditional approach to achieve such a decomposition is the introduction of a modularization abstraction. In our model, the above-mentioned attribute categories (from, to, by, decisionsemantic) characterize the interface of a conceptual decision module, whereas the "part" attribute not discussed so far characterizes the import interface of the decision module.†

In the planning phase of software development, modular decomposition is used for assigning system development work. In the usage phase of the information system, modular composition may be used for configuration management. A category of complex design decisions of particular interest to the DAIDA methodology are implementation hierarchies that relate a reasonably isolated world submodel, subsystem specification or conceptual design to its completed implementation. When generalized to a class defi-

†According to the DAIDA methodology, constraints at the CML level relate the implementation to the interface, and the parts to each other. Typically, the decomposition of a design decision is itself a design decision. This could be supported by AI-based planning and scheduling tools, also considering the goals of the design in a decision support setting [13]. The implementation of a module from the imported pieces is only characterized by constraints since the CML model just modularizes the requirements; typically, a TaxisDL script would be used to design the actual implementation.

nition by introducing parameters [48], such a component can be *reused* by re-instantiation; even incomplete hierarchies (e.g. requirements together with an associated design blueprint but no implementation) can be useful reusable objects [49].

In the following, we demonstrate the decomposition of design objects by introducing the complex decision class *mapandnormalizeInvitations* which aggregates the two decision instances introduced earlier. It takes as input the two TaxisDL entity classes *Papers* and *Invitations* and produces two *normalized* DBPL relations *InvitationRel_1* and *InvReceivRel* (see Section 4.1). The first part has already been done by mapping the TaxisDL design objects to a non-first-normal-form relation *InvitationRel_0*. The missing part is the mapping of *InvitationRel_0* to normalized relations. For this purpose we define a decision class *DBPL_RefNormalization* which models such mappings, and use this class for recording the normalization of *InvitationRel_0*:

```
INDIVIDUALCLASS DBPL_RefNormalization IN DesignDecision WITH
    from
        nonfirstrelations: DBPL_Rel_DO
    to
        normalizedrelations: DBPL_Rel_DO
    description
        normalizationdescr: NormDescription
END

INDIVIDUAL normalizeInvitations IN DBPL_RefNormalization WITH
    nonfirstrelations
        nfrel: InvitationRel_0
    normalizedrelations
        normrel1: InvitationRel_1
        normrel2: InvReceivRel
END
```

Finally, we aggregate the two parts to a complex decision class *StrucMapMoveDown*. The constraint expresses that for each instance, the part decisions must talk about the same objects as the complex one. One can easily see that it is fulfilled for the instance *mapandnormalizeInvitations*:

```
INDIVIDUALCLASS StrucMapMoveDown IN DesignDecision WITH
    from
        tdlentities: TDL_EC_DO
    to
        normalizedrelations: DBPL_Rel_DO
    part
        hiermap: EntHierMapMoveDown
        normalize: DPBL_RefNormalization
    constraint
        properdecomposition:
            $ hiermap.tdlentities = tdlentities &
              hiermap.nonfirstrelations = normalize.nonfirstrelations &
              normalize.normalizedrelations = normalizedrelations $
END
```

```
INDIVIDUAL mapandnormalizeInvitations
IN StrucMapMoveDown WITH
    tdlentities
        entity1: Papers
        entity2: Invitations
    normalizedrelations
        rel1: InvitationRel_1
        rel2: InvReceivRel
    hiermap
        step1: mapInvitations
    normalize
        step2: normalizeInvitations
END
```

The decomposition of design decision objects allows for the definition of complex methodologies, and reduces the size of dependency networks, combining ideas from programming-in-the-large (e.g. configuration management) with those for programming-in-the-small, such as constraint propagation for requirements or design modifications.

Note, that complex decision objects can also have descriptions, and thus dependencies relating their parts directly to each other, rather than having to go

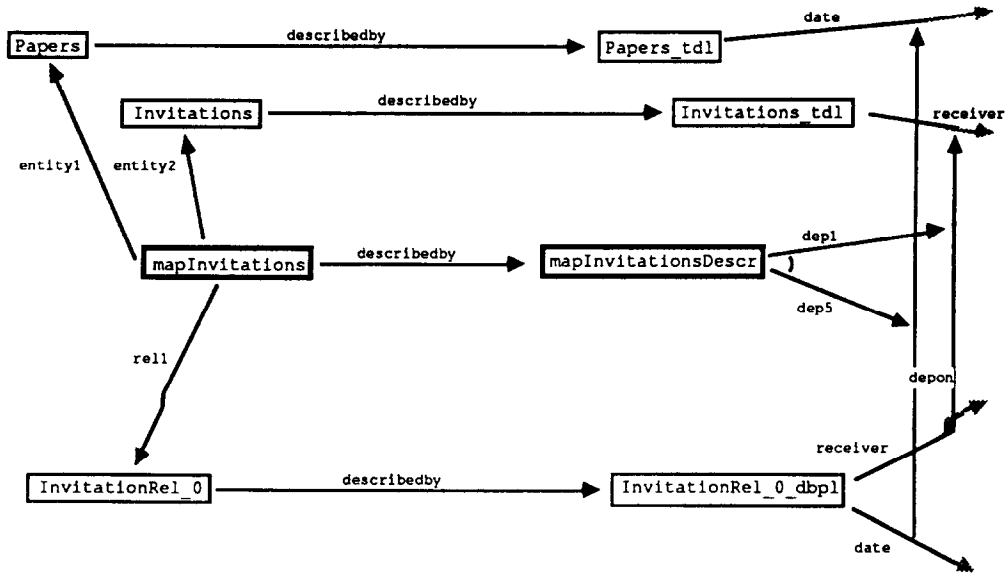

Fig. 15. Design object and decision modelling at the instance level.

via all subdecisions. In this way, design-in-the-large can use a derived, more compact dependency network for configuration, constraint propagation and search than the detailed recording of small-scale design decisions would allow. Another important advantage of the modularization is that decision classes can be used to define design-in-the-large methodologies such as the overall DAIDA methodology of decomposing the software development process in CML-based requirements analysis, CML–TaxisDL mapping, TaxisDL conceptual design, TaxisDL–DBPL program design and DBPL coding:†

```
INDIVIDUALCLASS DAIDAMethodology IN DesignDecision WITH
    from
        requirements: CML_DO
    to
        databaseprogram: DBPL_DO
    description
        implementationconstraint: DatabaseprogramSatisfiesRequirements
    by
        globaldaidenvironment: GKBMS
    part
        system_embedding: RequirementsAnalysis
        requirements_to_design: CML_TDL_Mapping
        design_consolidation: TDL_Integration
        design_to_program: TDL_DBPL_Refinement
END
```

Besides the vertical aggregation of decisions to development histories (at the instance level) respectively methodologies (at the class level), we also need a horizontal configuration, composing design objects and decisions from smaller ones, respectively decomposing complex tasks into more manageable ones. McMenamis and Palmer [52] provide some guidelines of how to do this (e.g. event based or data centered partitioning).

For example, when talking about the mapping of the generalization hierarchy of *Papers* and *Invitations*, we may wish to view this hierarchy as a single complex object, used as an input to a common decision. If more than one relation should result from the first subdecision (e.g. with the distribute strategy to use one relation per class), normalization could be performed in two separate subdecisions for the next step. An extension currently under development handles not only this case but also addresses the question of source configuration management, i.e. what happens if the desired conceptual configuration of objects does not coincide with physical file boundaries.

†In contrast to standard modularization approaches, however, it may be necessary to have multiple modularizations (or views) of the same structure; a deep discussion of the problems associated with such a multiple-viewpoint mechanism, often intended to support group work, is beyond the scope of this paper [50, 5].

4.5. Design tool modelling

If all software were developed by the DAIDA methodology, a design tool would be simply a reusable implementation hierarchy to be described at the levels of its CML systems requirements, TaxisDL conceptual design, implementation in some programming language, and possibly executable object code (derived automatically by compilation and thus not shown in Fig. 1).

At the CML level, the requirements of a tool are those of the design decision the tool is supposed to automate, typically a subdecision expected to occur in many design tasks. Thus, the class structure of design decisions can be used for describing the requirements of design tools. At the TaxisDL level, simple tools would be designed as transactions, whereas more complex ones would be specified as scripts for interactive problem-solving. In DAIDA, the CML–TaxisDL mapping assistant would help in generating these kinds of designs [9, 22]; the TaxisDL specification could also serve as a user guide through a complex tool.

In a real environment, of course, we wish to integrate pre-existing tools written in any programming language, as well as to develop new ones. We therefore have to construct CML and TaxisDL "envelopes" to make such tools known to the GKBMS (cf. [30] for the concept of envelopes in tool integration for software environments). The interaction with such tools can then be accomplished in several ways: a purely documentative one in which the user is just given information about the tool and then invokes it manually; an embedded procedure-call mechanism as in active databases (e.g. Postgres [62]); or a distributed message-passing protocol where GKBMS and tools are communicating active objects [53]. The current implementation only supports the first one while the second one is being implemented for the second prototype.

Of course, we assume that it has been established during the tool development process that the "to" object is a correct and complete implementation of the "from" object, i.e. that the tool does what it promises. Moreover, the description of any design tool relates the "from/to" parameters of the "from" *DesignDecisionObject* to the interface parameters of the called procedure, thus clarifying the meaning of these parameters in terms of the tool requirements. Note that, while *ExecutableProcedureCalls* basically introduce the active database functionality provided by object-oriented languages such as SmallTalk [31], the GKBMS approach embeds the use of these methods in the pre-/postcondition controls defined by the calling decision classes to provide some knowledge about the semantics of the methods. This also defines something like (nested) design transactions.

Instances of *DesignTool* are specifications of tools available in a concrete software engineering environment. The corresponding tool objects normally have system-generated identifiers; therefore, we allow to substitute some surface representation of the procedure call in the same way we introduced the "!" notation for naming attribute objects implicitly. In fact, the user would normally only see these surface representations while the input–output information would be internal information generated and used by the system. This information hiding can be used to identify applicable tools in an efficient way by linking them physically directly to object classes (i.e. storing redundant derived information), or for other optimizations.

As an example, assume that the "mapping assistant" supporting the normalization sub-decision in Section 4.2 is a Prolog procedure whose highest level might be defined roughly as follows:

```
normalize ([], _).

normalize ([_firstel|_restinput], [_firstrel|_restoutput]):-
    hasnosetvaluedattr (_firstrel),
    normalize (_restinput, _restoutput).

normalize ([_firstrel|_restinput], _restoutput):-
    hassetvaluedattr (_attrlist1, _firstrel),
    haskey (_atrlist2, _firstrel),
    formrel (_attrlist2, _attrlist1, _newrel),
    append (_newrel, _restinput, _newrestinput),
    subtractattributes (_firstrel, _attrlist1, _firstrelreduced),
    normalize ([_firstrelreduced|_newrestinput], _restoutput).
```

The corresponding tool object might look like this:

```
INDIVIDUALCLASS $normalize (nonfirstrelations, normalizedrelations)$
IN DesignTool WITH
    from
        toolspec: DBPL_RefNormalization
    to
        toolexec: PrologCall
END
```

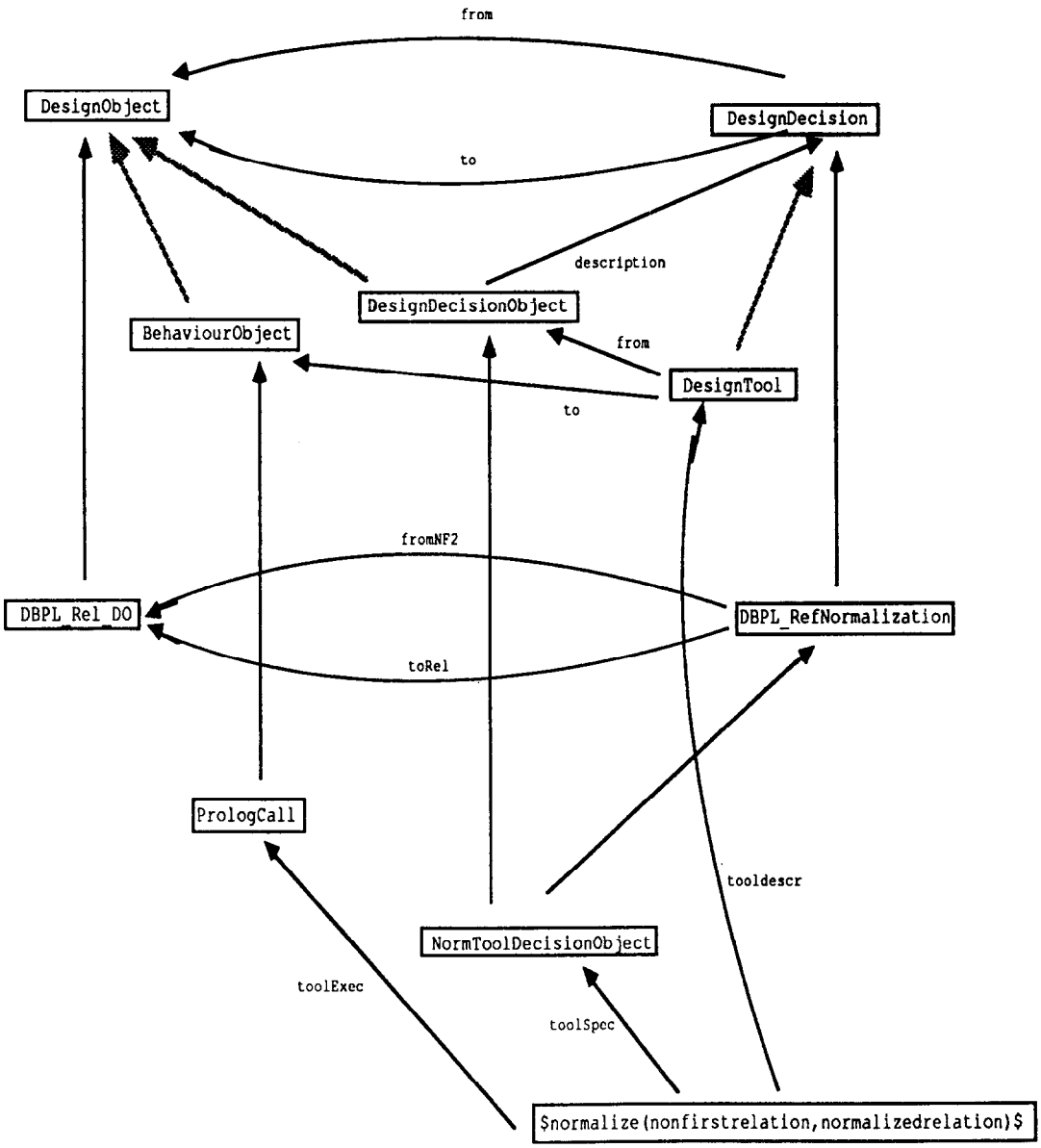

Fig. 16. Tool embedding in the GKBMS software process model.

Figure 16 gives the semantic network structure for this example. This tool model is also used to describe the tools for the ConceptBase environment itself, in particular, the user interface tools, the inference engines and consistency checkers for rule and constraint processing, and the secondary storage management (see Section 6.3).

5. CONCEPTBASE: A PROTOTYPE IMPLEMENTATION

Like other data-intensive information systems constructed with the DAIDA approach, the software process data model should be realized with the DAIDA tools sketched in Section 2.1. However, since these are far from completed and would themselves need support from the GKBMS, the initial GKBMS implementation is based on a simpler support system named ***ConceptBase*** (Conceptual Model Base Management System) from which more efficient implementations for very large knowledge bases will be bootstrapped.

ConceptBase implements a CML kernel and usage environment based on the definitions in Section 3, augmented with features to describe multiple views of knowledge, system behaviours, complex object configurations and display facilities. This kernel can also serve as an implemented semantic specification for other implementations. A first prototype has been operational since spring 1988 [45]; a second one is scheduled for completion in April, 1989 [64]. The system runs on SUN-Workstations under Unix and

currently (February 1989) comprises about 40,000 lines of BIM-Prolog, C and interface code; the second prototype also runs on VAX under VMS.

The ConceptBase architecture, shown in Fig. 17, follows the three language levels of network, frame and conceptual model, offering extensibility and optimization strategies at each level to achieve efficiency. In the figure, strong boxes indicate modules which have been implemented and integrated into the system, whereas dotted boxes indicate modules either not yet integrated or not even fully implemented. Our software process data model can be considered one particular conceptual model; others, e.g. for team support (design conversation base) are being studied.

5.1. The ConceptBase kernel system

The interface of the ***Proposition Processor*** represents CML propositions at the network level by Prolog 5-tuples:

propval (id, source, label, destination, interval).

which are internally further subdivided and represented as a knowledge base graph with efficient main memory-oriented database access. To work on these objects, three operations are provided:

- create_proposition(_p)—create the proposition _p in the knowledge base,
- retrieve_proposition(_p)—search for a proposition matching _p,
- store_proposition(_p)—create _p if not already existent and
- delete_proposition(_p)—delete the proposition _p.

The client of the proposition processor, the ***Object Processor***, configures sets of propositions according to certain criteria, usually around a common source to build a *frame*. A frame object is internally represented as a CML-fragment which resembles the parse tree of the frame-level syntax; the exact translations between frames and fragments, and between fragments and propositions is described in [45]. The tell and ask operations of the frame-level interface are translated to corresponding updates and queries at

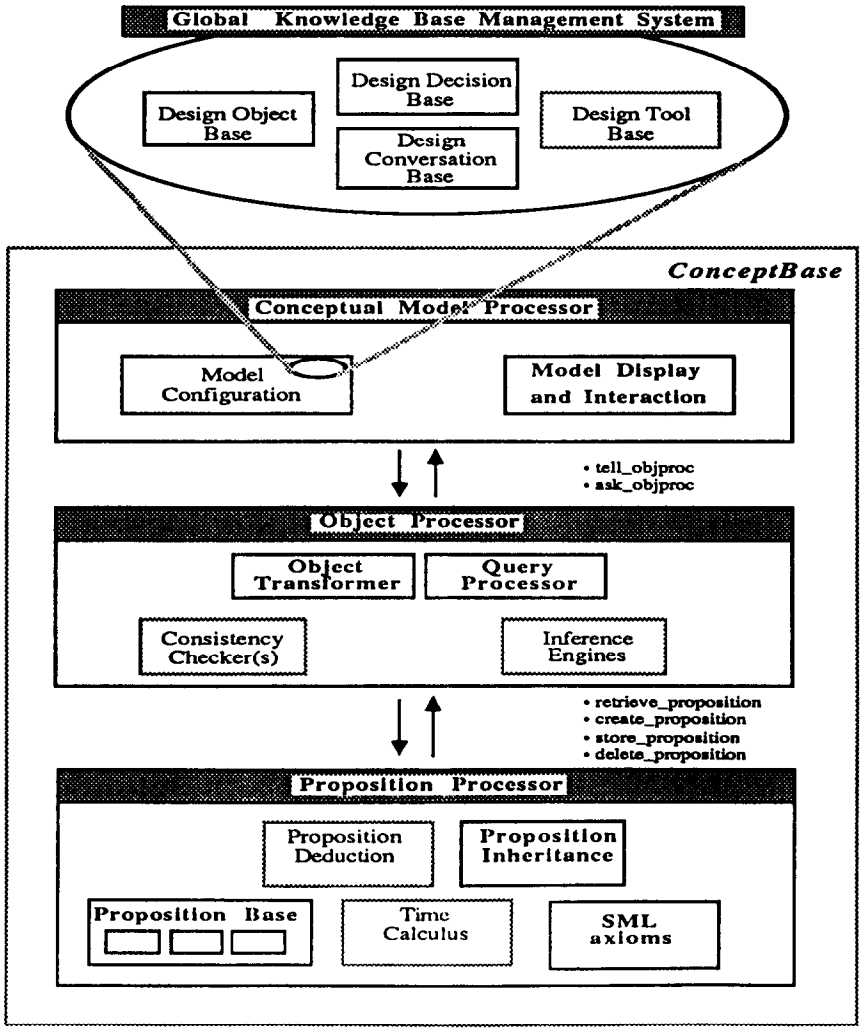

Fig. 17. ConceptBase architecture and implementation status.

the fragment level. The feasibility of an object-level update transaction is verified by the ***Consistency Checker*** which utilizes information of the proposition processor. A special feature of ConceptBase, pioneered by the KRYPTON system [46], is that the consistency checker has to integrate several kinds of integrity checking: enforcing the above-mentioned CML axioms, taking into account *temporal* consistency, and supporting one or more *predicative* assertion languages (subclasses of attribute class "constraint"). Recently proposed simplification algorithms for deductive databases (e.g. [54, 39]) only support the assertional part of this problem; since a whole set of operations may be passed to the proposition processor together, set-oriented optimization of the consistency check is being studied.

The ***Inference Engines*** may support various proof strategies for querying object properties via first-order logic expressions over CML objects. Since the same assertion language is used in rules (see *rule* propositions above), the inference engines are also capable of evaluating deduction rules. Several time calculi, e.g. Allen's interval calculus [40] may be supported as well.

In the first prototype, the ***Query Processor*** is mostly geared towards a focusing/browsing style of search; the second prototype also contains full rule-based querying facilities. The interface is implemented by the operation, ask_objproc(_q,_a), where _q stands for the query and _a for the answer. Possible values for _q are:

exists(_x)
: The answer is "yes" if there is an object with identifier _x in the proposition processor.

get_object(_x)
: Information connected to _x is collected and returned as a frame data structure (called CML-fragment).

get_links(. . .), get_ids(. . .)
: A list of connected links (nodes) with common properties is computed and returned.

[each, _pattern, where, _l1. . . . , _ln]
: The answer contains all terms matching _pattern which satisfy the conjunction of the literals _l1, . . . , _ln.

The second operation of the object processor, tell_objproc(_i,_r), passes new information to it. The parameter _i contains the information as a list of CML-fragments. If there are no syntactic or semantic errors, the object transformer translates the information into a set of equivalent propositions which is stored in the proposition processor and returned in parameter _r. Otherwise, _r holds the value "error".

5.2. The ConceptBase usage environment

The ConceptBase usage environment is intended to make the hypertext-like style of CML practically available to the user. As a consequence, browsing, viewing and editing of knowledge bases should be possible symmetrically on the network as well as on the textual frame representation. In a typical knowledge engineering process for information systems development, an initial sketch of the knowledge base is obtained with graphical tools, then the details are worked out using textual tools.

Formally, the interface tools are tools as described in Section 4.4, relating the content of the knowledge base to a (screen) view of it, according to a view definition that characterizes both the content and the layout of the view. By restricting the possible view definitions, most views can be made updatable; moreover, to gain different perspectives on the software process knowledge base, different symbols can be associated with objects of particular classes, thus mimicking well-known representational views such as data flow diagrams, entity–relationship diagrams, etc. In the following, we give a brief overview of the tools that are available for the current prototype [45].

The ***Conceptual Model Processor*** uses the object processor to combine tools for the manipulation of models which consist of all objects relevant to an application of ConceptBase, e.g. the GKBMS. Models constitute highly complex multi-level object structures which are maintained in hierarchies. Different models may share some objects or (sub-)models. Configuring a model for a specific application means the activation of the corresponding nodes in the lattice, i.e. making their objects accessible for the proposition processor. This work is done by the ***Model Configuration*** module which corresponds to a complex object database; to date, only a simple main memory version of this component has been implemented.

The ***Display and Interaction*** module integrates man–machine communication into ConceptBase objects and models; individual frame objects can be displayed and modified interactively, and models can be displayed, browsed and possibly reorganized in textual and graphical style.

For the sake of modularity, the display and interaction module is implemented in two layers. The bottom layer provides a set of *interface tools* which process uninterpreted strings (e.g. object identifiers) and structures; these interface tools do not know anything about the semantics of displayed objects and structures. The *usage environment* relates these interface tools to the object processor by requesting object identifiers to be used in the interface tools. The current ConceptBase prototype offers the following interface development tools:

- declaration of menus and associated tools;
- textual and graphical editing of CML objects with syntactic and semantic checking;
- relational display with selection facilities;
- textual and graphical browsing of tree-like structures (also with selection);

- interaction to obtain text commands from a user;
- error window to record and display error messages of ConceptBase.

These tools are embedded in a usage environment accessible through the ***ConceptBaseToolBar***, which itself is realized by the menu declaration tool. Three main kinds of interaction with the knowledge base are currently offered:

- textual browsing of user-defined sub-networks (TextBrowser),
- graphical browsing of user-defined sub-networks (GraphBrowser),
- syntactically and semantically controlled object display and update (Editor).

Additionally, a *system menu* offers internal system operations (bulk-loading CML objects stored on external files, executing Prolog calls and stopping the system) and a *configuration menu* supports composition of conceptual models from submodels (invoking the *Model Configuration* module).

The ***TextBrowser*** queries the user for a specification of the structure to be browsed by calling the interaction tool. Basically, such a specification consists of two parts. The first one specifies the *focus*, i.e. the root of the hierarchical structure. The other one specifies how to compute the lower levels. The latter specifications are founded on the net-like representation of CML in the PropositionProcessor, but accessed through the *get_ids* operation of the object processor. After completing the system, we noted its similarity to recent, independently developed so-called "idea processors" which allow a user to play with different alternative organizations for texts [55].

Similarly, but using the *get_links* rather than the *get_ids* operation, the ***GraphBrowser*** obtains a net-like specification by calling the interaction tool, computes the corresponding structure of object identifiers using the object processor, and passes this structure to the graphical browsing tool.

Both browsers permit the selection of objects, and invocation of tools such as the editor. The ***Editor*** allows displaying, analyzing, modifying and creating CML objects. Scanning, parsing and transformation to CML-fragments is performed by Prolog programs automatically generated from definite clause gram-

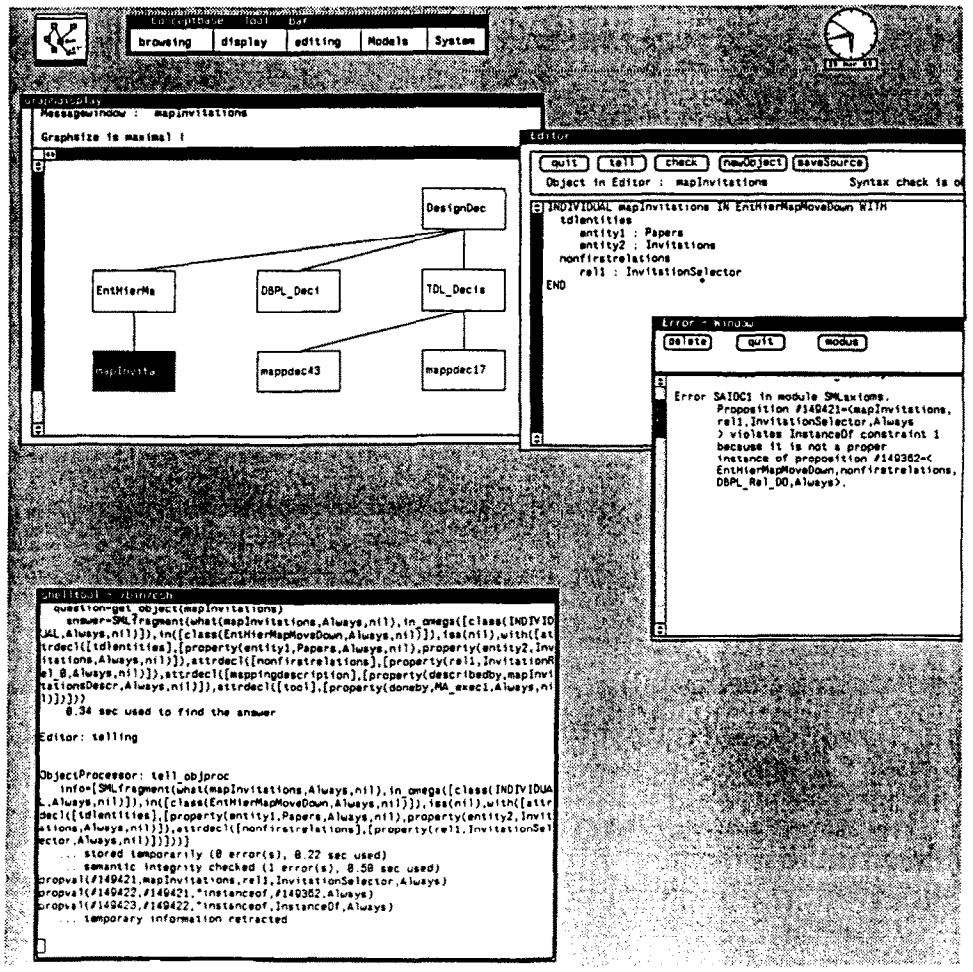

Fig. 18. Interaction of ConceptBase kernel and usage environment (mapping example).

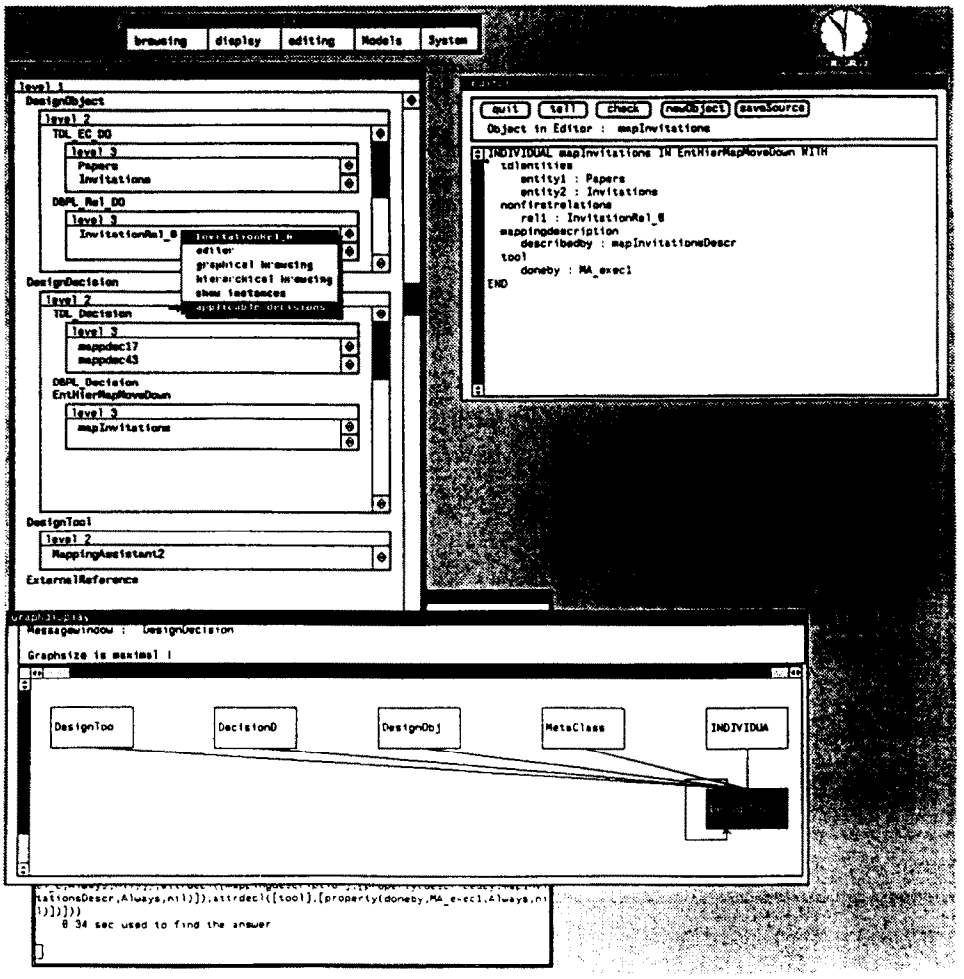

Fig. 19. Interaction of ConceptBase kernel and usage environment (hierarchical browsing).

mars [56]. Thus, the editor can be easily adapted to a modified syntax or ObjectProcessor interface. Semantic integrity is checked by the ObjectProcessor during the *tell* operation. Each detected error is reported to an error window.

The screendump in Fig. 18 illustrates the interaction between (graphical) browser, editor and ObjectProcessor, using a small subproblem from the mapping example in Section 2.2. First, the user invoked the *GraphBrowser* to display all instances of DesignDecision and all instances of these instances (the object *mapInvitations* is an instance of *EntHierMapMoveDown* which is an instance of *DesignDecision*, cf. Fig. 10). The user query was transformed into an appropriate call of *ask_objproc* returning a list of edges ready for layout by the graph browser.

In the next step, the user mouse-selected the *mapInvitations* node, and chose the editor tool from the displayed menu to zoom into and document the execution of this design decision (cf. also Fig. 3 and Section 2.3). The editor obtained the object frame (as known before the execution of the decision) by asking the ObjectProcessor for the existence of *mapInvitations* and, since it existed, for the corresponding CML fragment (shown at the top of the session protocol in the "shelltool" window). Then, the user added the output attribute for *mapInvitations* and pressed the "tell" button. After successful parsing (shown in the upper part of the editor window), the corresponding CML-fragment was passed to the Object Processor which stored it temporarily and checked the structural integrity of the new information. In this example, an error was detected and reported in the error window: attribute "rel1" does not match its category "nonfirstrelations" since the design object InviteSelector is not an instance of DBPL_Rel_DO (it represents a DBPL selector rather than a DBPL relation).

Subsequently, the screen dump in Fig. 19 demonstrates the use of the hierarchical TextBrowser for obtaining an overview of the work done so far. It shows the situation after the first sub-decision of our example; the pop-up menu option "applicable decisions" is just being activated, ostensibly leading to the second sub-decision (normalization).

6. APPLICATIONS

The software process data model exploits the combination of the design decision idea and object-oriented construction principles to offer sufficient extensibility so that not only new tools but also new theories can be continuously added to the environment and can be made reusable with little effort. Most importantly, of course, this should apply to the knowledge-based development support theories and tools developed in other subprojects of DAIDA. At least for the two mapping tasks from SML to TaxisDL, and from TaxisDL to DBPL, as well as for the requirements analysis task within SML, experiments have already started to classify and formalize these sub-environments so that they can utilize the GKBMS fully. Additionally, we are using the model extensively in the design and implementation of the ConceptBase system itself.

6.1. Requirements modelling and design mapping

CML and TaxisDL are formally rather similar languages, however, with different tasks in the DAIDA methodology. The CML level is concerned with collecting and organizing the requirements for the system to be developed. In doing so, it also has an important function in steering the subsequent design process, especially by considering *design goals* which can later be used for helping users choose among applicable decision classes [13]. So far, DAIDA has mostly considered *functional* goals as the driving force for the decision classes (this is also what the first ConceptBase prototype supports) while other goals (performance, modularity, ...) were at best treated as constraints or only as comments. Recently, experiments with integrating goal-oriented multiple criteria decision support into the model have begun [57].

Within the requirements level, decisions have to be made what views of the world model to represent in the system model. Assuming this has been done, the CML–TaxisDL mapping [22] then decides how to represent the system model specification in TaxisDL terms, especially considering how much to represent the system model specification in TaxisDL terms, especially considering how much of the historical information present in the CML model should be retained for the TaxisDL model. Furthermore, class hierarchies can be reorganized with a view on efficient implementation, e.g. defining a new subclass for current information and storing the rest in another subclass that the TaxisDL–DBPL mapping could then relegate to a slow storage medium.

A first attempt at classifying the kinds of decision classes to be made at these levels has given rise to the hope that an orthogonal combination of the following two kinds of decision classes could represent a structured and fairly complete coverage:

- ***Ontology***—Design objects at both levels come as informations about either *entities*, *activities*, *constraints* or *goals*. Thus, we need classes for: (a) developing requirements for these; (b) deciding which of them to represent in the system; and (c) to what degree and with what methods (especially concerning time) to map them between CML and TaxisDL. The choice between the possible decisions should be governed by the design goals specified in the requirements analysis.
- ***Epistemology***—CML and TaxisDL provide (slightly different versions of) abstraction principles like *aggregation*, *generalization* and *classifications*, together with their reverse operations of *decomposition*, *specialization* and *instantiation*. Each of these six abstraction (resp. specification) operators corresponds to a decision class that specifies a relationship between smaller and larger objects or subtasks. For example, aggregation can be used to relate the mapping of a whole class to the mapping of its attributes; similarly, mapping of *IsA* relates the mapping of a complete hierarchy of objects (as in our TaxisDL–DBPL example) to that of its individual members. Goal decomposition as a strategy for elaborating requirements within the CML level is another example of an aggregation class, whereas (as in our software process model definition) classification can be provided to define suitable application-specific sublanguages for a mapping task. Note that classification differs between CML and TaxisDL: a CML metaclass heirarchy has to be flattened in the mapping to TaxisDL, using metalevel amalgamation similar to the one proposed in [58].

Ideally, there should only be a small set of basic mapping decisions for each of the above types, rather than separate rules for all conceivable combinations or even sequences of combinations. Using orthogonal aggregation of such decision classes, more complex methodologies for the mapping can be formed. This would clarify the structure of dependencies at the description level as well as facilitating communication between the individual tools and the GKBMS.

6.2. TaxisDL–DBPL mapping

In the examples of this paper, the mapping task from the object-oriented knowledge representation language TaxisDL to the set-based, module-oriented database programming language DBPL has been highly oversimplified. Indeed, we only considered some of the data structure aspects; the mapping of transactions turns out to be much more difficult and requires full support by formal software development methods. The method used in DAIDA exploits experience with mathematical specification techniques, using the language Z and its derivatives [21]. In this approach, design objects correspond to so-called abstract machines that represent data structures, operations and constraints of a particular application module; decisions correspond to formal transformations supported by theorem-proving assistance tools.

Based on these experiences, the TaxisDL–DBPL mapping is intended to proceed in three steps with corresponding decision classes [24]:

- translation of TaxisDL model to abstract machine à la Abrial,
- refinement of abstract machine towards efficient, modular implementation,
- translation of final machines to DBPL program.

Disregarding the initial and final steps (which are automated translations), the intermediate design *objects* are abstract machines whose descriptions have roughly the following structure:

```
INDIVIDUALCLASS AbstractMachine
IN DesignObject WITH
attribute
    context: DataObjects
    variable: Name
    invariant: FunctionalConstraintClass
    operations: FunctionText
END
```

The *decision* classes of this mapping correspond to generalized substitutions in abstract machines; in contrast to the CML–TaxisDL mapping, such substitutions consider entity, activity and constraint mapping simultaneously. Among the abstraction operations mentioned above, aggregation of such objects plays the central role. There is no generalization (although the notion of substitutions is closely related to that of inheritance) while metaclass-like notation extensions are simulated by import from other abstract machines. An important aspect of decision semantics in the sense of our model is the documentation and management of proof obligations and already proven lemmata.

6.3. ConceptBase development

The software process data model has also played a major role in designing and implementing the ConceptBase system itself. The main emphasis has been on dealing with very large software knowledge bases, and on providing multiple views with user-friendly interaction facilities in a uniform framework. In [44], three specific application areas are described in detail.

Efficient deductive query processing and integrity checking—CML rules and constraints are modelled internally as particular (deterministic†) decision classes for which tools—triggered query processors and constraint checkers—are automatically generated by tools associated with the predefined metaclasses *RuleClass* and *ConstraintClass*. Luckily, the decision class structure turns out to provide exactly the kind of graphs needed for the plethora of algorithms proposed for deductive query optimization [60] and integrity control [54, 39]. Specialized graph structures can be defined by specialized attribute categories for the input–output attributes. Thus, the structure is independent of a particular style of rule or optimization algorithm; specific optimization ideas can be defined at the metalevel as in rule-based optimizer generators, thus serving as a testbed for various optimization procedures. An extension of the algorithm in [54] is currently being integrated into the second ConceptBase prototype [61]. Note that, using redundant design object and design decision classes together with the dependency structures defined in their descriptions, we can also integrate the redundant storage and maintenance of derived data to increase efficiency.

Version and configuration management—Configurations are viewed as composite objects put together according to configuration decisions. The use of the decision-based version and configuration model has substantially simplified the portation of the initial SUN-UNIX prototype to the VAX-VMS version. Commercial configuration tools such as MAKE in UNIX or MMS in VMS support such decisions at the source level and administer the ConceptBase system components (currently about 80 system modules, plus many example applications). In combination with a conceptual configuration decision model under development in our group, version and configuration management will become possible even across heterogeneous hardware and system software environments [63].

Knowledge base perspectives and user interfaces—The above models can be applied to the handling of multi-window interactions with the system in a hypertext-like style. A window is viewed as a particular configuration of derived objects which corresponds to a configuration of internal knowledge base objects, thus giving a clean semantics to window-based updates. For this purpose, the configuration model had to be extended by equivalent representation mapping decisions.

Summarizing, the software process data model provides us with a way to describe a large number of important implementation issues not just with obscure internal languages but with the surface knowledge representation language of the system itself, thus facilitating experimentation with, and extensibility of, the system.

7. CONCLUSIONS

In this paper, we proposed a data model which represents software development as a process of *tool*-supported design *decisions* operating on abstract design *objects*. This model is different from other attempts in that it explicitly considers the functionality of tools, but at the same time emphasizes the

†There is an interesting relationship between the design decision concept in general with non-deterministic database update operations as discussed in [59]. This relationship could serve as the foundation of a theory of the power of particular design decision class languages but we have hardly begun to study this idea.

non-deterministic nature of human design decisions. Moreover, the way *how* tools are attached to design decisions seems to point a way out of the integrity control problems associated with freely usable methods in some object-oriented languages and databases.

Although the experience with various experimental applications is quite encouraging, several extensions appear useful or even necessary.

Firstly, we would like to *broaden the scope of development paradigms* beyond the initial DAIDA approach. One alternative method, followed in the new ESPRIT project ITHACA, is to strengthen the emphasis on reusability beyond the context of tool modelling; based on a requirements model, existing building blocks are selected from a software library and configured to application systems, rather than developing new programs each time. Another alternative, currently being studied for environmental protection applications in collaboration with the FAW Institute in Ulm, West Germany, is the loose coupling of independently developed software systems under the common conceptual umbrella of a "competence model". Here, the idea is to make organizational knowledge available to users even if no coherent requirements analysis has been conducted.

The second group of extensions concerns more explicit *support for the decision-making process*. In particular, we wish to take seriously the *IsA* link between the metaclasses *DesignDecision* and *DesignObject* in our model, i.e. design decisions are objects that can evolve, be talked about, justified by other decisions, etc. On the one hand, this requires a better understanding of decision support methodologies for goal-driven design. On the other, we have to set up a design conversation network among the stakeholders and workers in a software project. This involves the conceptual representation of agents, structural messages, negotiation positions, commitments and the like, but also the introduction of group support tools such as multi-media real-time conferencing support. Corresponding extensions of our model and of the ConceptBase prototype are implemented in the second prototype [64].

A final set of research questions is concerned with *broadening the scope of application areas* to design and maintenance tasks beyond the information systems domain. Co-authoring of technical natural language documents (e.g. user documentation for software) is a typical candidate we are currently beginning to investigate [65].

Acknowledgements—This work was supported in part by the European Commission under ESPRIT Contract 892 (DAIDA) and by the Deutsche Forschungsgemeinschaft in the "Objectbanks for Experts" program (Grant Ja445/1-1). DAIDA partners include the software houses BIM/Belgium, GFI/France, SCS/Germany; and the research institutions FORTH/Greece, University of Frankfurt/Germany and University of Passau/Germany. The authors are grateful to the other DAIDA partners for valuable discussions, and to many students, notably Michael Gocek, Eva Krüger, Hans Nissen and Martin Staudt for implementation work. Comments by John Mylopoulos helped to put our work better in perspective.

REFERENCES

[1] M. L. Brodie and J. Mylopoulos (Eds). *On Knowledge Base Management Systems*. Springer-Verlag, New York (1986).

[2] V. Dhar and M. Jarke. Dependency-directed reasoning and learning in systems maintenance support. *IEEE Trans. Software Engng* **SE-14**(2), 211–227 (1988).

[3] J. Mylopoulos, P. A. Bernstein and H. K. T. Wong. A language for designing interactive data-intensive applications. *ACM Trans. Database Systems* **5**(2), 185–207 (1980).

[4] S. Greenspan, A. Borgida and J. Mylopoulos. A requirements modelling language and its logic. In [1] pp. 471–502 (1986).

[5] J. Winkler (Ed.). *Proc. Int Workshop on Software Versioning and Configuration Control*, Grassau, F.R.G. Teubner, Stuttgart, F.R.G. (1988).

[6] P. A. Bernstein. Database support for software engineering. *Proc. 9th Int. Conf. Software Engineering*, San Francisco, pp. 166–178 (1987).

[7] L. A. Rowe and S. Wensel (Eds). *Proc. ACM-SIGMOD Workshop on Software CAD Databases*, Napa, Calif. (1989).

[8] A. Borgida, M. Jarke, J. Mylopoulos, J. W. Schmidt and Y. Vassiliou. The software development environment as a knowledge base management system. *Foundations of Knowledge Base Management* (Edited by J. W. Schmidt and C. Thanos). Springer-Verlag, Heidelberg (1989).

[9] M. Jarke. DAIDA Team. The DAIDA environment for knowledge-based information systems development. *Proc. ESPRIT Conf. '88: Putting the Technology to Use*, Brussels, Belgium, pp. 405–422 (1988).

[10] P. P. S. Chen. The entity–relationship model: towards a unified view of data. *ACM Trans. Database Systems* **1**(1), 9–36 (1976).

[11] J. L. Peterson. Petri nets. *ACM Comput. Surv.* **9**(3), 223–252 (1977).

[12] J. Doyle. A truth maintenance system. *Artificial Intell.* **12**, 231–272 (1979).

[13] J. Mostow. Towards better models of the design process. *AI Mag.* **6**(1), 44–57 (1985).

[14] M. T. Stanley. CML: a knowledge representation language with application to requirements modeling. M.S. Thesis, University of Toronto, Canada (1986).

[15] M. Koubarakis, J. Mylopoulos, M. Stanley and M. Jarke. Telos: a knowledge representation language for requirements modelling. Technical Report CSRI-222, University of Toronto (1988).

[16] A. Borgida, E. Meirlaen, J. Mylopoulos and J. W. Schmidt. The TAXIS design language (TDL). Report, ESPRIT Project 892 (DAIDA), Institute of Computer Science, Research Center of Crete, Greece (1987).

[17] J. W. Schmidt, H. Eckhardt and F. Matthes. Extensions to DBPL: towards a type-complete database programming language. Report ESPRIT Project 892 (DAIDA), Universität Frankfurt, F.R.G. (1988).

[18] F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, P. Pepper, K. Samuelson, M. Wirsing and H. Wössner. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L, Lecture Notes in Computer Science 183*. Springer-Verlag, Berlin (1985).

[19] D. R. Smith, G. B. Kotik and S. J. Westfold. Research

on knowledge-based software engineering environments at Kestrel Institute. *IEEE Trans. Software Engng* **SE-11**(11), 1278–1295 (1985).

[20] R. C. Waters. The programmer's apprentice: a session with KBEmacs. *IEEE Trans. Software Engng* **SE-11**(11), 1296–1320 (1985).

[21] J. M. Spivey. An introduction to Z and formal specifications. Tutorial Notes, Oxford University, U.K., Presented at *ESEC '87*, Strasbourg, France (1987).

[22] M. Mamalaki, M. Marakakis, M. Mertikas, T. Topaloglou and Y. Vassiliou. On the development of information systems: from requirements modelling to system design. *Proc. EURINFO '88*, Athens, pp. 560–567 (1988).

[23] E. Meirlaen and J.-M. Trinon. An object-based prototyping workbench for Prolog. *Proc. ESPRIT Conf. '88: Putting the Technology to Use*, Brussels, Belgium, pp. 423–437 (1988).

[24] M. Weigele and I. Wetzel. TDL-DBPL mapping: methodology and first experiences. Report, ESPRIT Project 892 (DAIDA), Universität Frankfurt, F.R.G. (1988).

[25] D. Batory and W. Kim. Modeling concepts for VLSI CAD objects. *ACM Trans. Database Systems* **10**(3), 322–346 (1985).

[26] M. Jarke, M. Jeusfeld and T. Rose. A global KBMS for database software evolution: design and development strategy. Report MIP-8722, Universität Passau, F.R.G. (1987).

[27] M. Bouzeghoub, G. Gardarin and E. Metais. Database design tools: an expert systems approach *Proc. 11th Int. Conf. Very Large Data Bases*, Stockholm, Sweden, pp. 82–95 (1985).

[28] G. E. C. Weddell. Physical design and query compilation for a semantic data model. Ph.D. Thesis, Dept Computer Science, University of Toronto (1987).

[29] M. Jarke, V. Linnemann and J. W. Schmidt. Data constructors: on the integration of rules and relations. *Proc. 11th Int. Conf. Very Large Data Bases*, Stockholm, pp. 227–240 (1985).

[30] S. A. Dart, R. J. Ellison, P. Feiler and N. Habermann. Software development environments. *IEEE Comput.* **20**(11), 18–28 (1988).

[31] A. Goldberg and D. Robson. *SMALLTALK 80: The Language and its Implementation*. Addison Wesley, Reading, Mass. (1983).

[32] R. Katz, E. Chang and R. Bhateja. Version modeling concepts for computer-aided design databases. *Proc. SIGMOD Int. Conf. Management of Data*, Washington, pp. 379–386 (1986).

[33] K. Abramowicz, K. R. Dittrich, W. Gotthard, R. Längle, P. C. Lockemann, T. Raupp, S. Rehm and T. Wenner. Datenbankunterstützung für Software-Produktionsumgebungen. *Proc. Datenbanken in Büro, Technik und Wissenschaft*, Darmstadt, F.R.G., pp. 116–131 (1987).

[34] J. M. Smith and D. C. P. Smith. Database abstraction: aggregation and generalization. *ACM Trans. Database Systems* **2**(2), 105–133 (1977).

[35] P. Lyngbaek and W. Kent. A data modelling methodology for the design and implementation of information systems. *Proc. Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, Calif. (1986).

[36] S. E. Hudson and R. King. Object-oriented database support for software engineering. *Proc. ACM-SIGMOD Int. Conf. Management of Data*, San Francisco, pp. 491–503 (1987).

[37] J. Banerjee, W. Kim, H.-J. Kim and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *Proc. ACM-SIGMOD Int., Conf. Management of Data*, San Francisco, pp. 311–322 (1987).

[38] J. Gray. The transaction concept: virtues and limitations. *Proc. 7th Int. Conf. Very Large Data Bases*, Cannes, pp. 144–154 (1981).

[39] F. Sadri and R. A. Kowalski. A theorem-proving approach to database integrity checking. *Foundations of Deductive Databases and Logic Programming* (Edited by J. Minker), pp. 313–362. Morgan Kaufmann, Los Altos, Calif. (1988).

[40] J. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM* **26**(11), 832–843 (1983).

[41] R. G. Smith, T. M. Mitchell, P. H. Winston and B. G. Buchanan. Representation and use of explicit justifications for knowledge base refinement. *Proc. 9th Int. Joint Conf. Artificial Intelligence*, Los Angeles, pp. 673–680 (1985).

[42] A. Borgida, T. Mitchell and K. Williamson. Learning improved integrity constraints and schemas from exceptions in databases and knowledge bases. In [1] pp. 259–286 (1985).

[43] M. Koubarakis, J. Mylopoulos, M. Stanley and A. Borgida. Telos: features and formalization. Technical Report FORTH/CSI/TR/1989/018, Computer Science Institute, Iraklion, Greece (1989).

[44] M. Jarke, M. Jeusfeld and M. Rose. Software process modelling as a strategy for KBMS implementation. *Proc. First Int. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan, pp. 495–515 (1989).

[45] M. Jarke, M. Jeusfeld and T. Rose. A global KBMS for database software evolution: implementation of first ConceptBase prototype. Report MIP-8819, Universität Passau, F.R.G. (1988).

[46] R. J. Brachman and H. Levesque. Tales from the far side of KRYPTON. *Proc. First Int. Conf. Expert Database Systems*, pp. 3–43. Benjamin Cummings, Menlo Park, Calif. (1987).

[47] J. de Kleer. An assumption-based TMS. *Artificial Intell.* **28**(2), 127–163 (1986).

[48] F. Bancilhon. Object-oriented databases. *Proc. 7th ACM Symp. Principles of Database Systems*, Austin, pp. 152–163 (1988).

[49] S. Katz, C. A. Richter and K.-S. The. PARIS: a system for reusing partially interpreted schemata. *Proc. 9th Int. Conf. Software Engng.* Monterey, pp. 377–386 (1987).

[50] G. Attardi and M. Simi. Metalanguage and reasoning across viewpoints. *Proc. ECAI '84*, Pisa, Italy, pp. 315–324 (1984).

[51] D. S. Wile and D. G. Allard. Worlds: an organizing structure for object-bases. *Proc. 2nd Symp. on Practical Software Environments* (1986).

[52] S. M. McMenamis and J. F. Palmer. *Essential Systems Analysis*. Yourdon Press, Englewood Cliffs, New Jersey (1984).

[53] D. Tsichritzis (Ed.). Active object environments. Report, Centre Universitaire d'Informatique, Université de Geneve, Switzerland (1988).

[54] F. Bry, H. Dekker and M. Manthey. A uniform approach to constraint satisfaction and constraint satisfiability in deductive databases. *Proc. EDBT*, Venice, Italy, pp. 488–505 (1988).

[55] L. F. Young. *Decision Support and Idea Processing Systems*. Dubuque. Brown, Iowa (1988).

[56] F. C. N. Pereira and D. H. D. Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial Intell.* **13**, 231–278 (1980).

[57] L. Pletz. Mehrkriterienunterstützung für Entwurfsentscheidungen in Softwareprozessen. Diploma Thesis, University of Passau, F.R.G. (1989).

[58] K. A. Bowen and R. A. Kowalski. Amalgamating language and metalanguage in logic programming. *Logic Programming* (Edited by S. A. Tärnlund), pp. 153–172. Academic Press, New York (1982).

[59] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. INRIA Research Report 900, Rocquencourt, France (1988).

[60] J. D. Ullman. *Principles of Database and Knowledge-base Systems*, Vol. 2. Computer Science Press, Rockville (1989).

[61] E. Krüger. Integritätsprüfung in deduktiven Objektbanken am Beispiel von ConceptBase. Diploma Thesis, University of Passau, F.R.G. (1989).

[62] T. Rose and M. Jarke. A decision-based configuration process model. *Proc. 12th Int. Conf. on Software Engng*, Nice, France (1990).

[63] S. Eherer, M. Jarke, M. Jeusfeld, A. Miethsam and T. Rose. *ConceptBase V2.0 User Manual*. Report MIP-8936, University of Passau, F.R.G. (1989).

[64] U. Mahn, M. Jarke, K. Kreplin, M. Farusi and F. Pimpinelli. CoAUTHOR: a hypermedia group authoring environment. *Proc. European Conf. on Computer-Supported Cooperative Work*, Gatwick, U.K. (1989).